BPFContain: Towards Secure and Usable Containers with eBPF

COMP5900I Final Project

William Findlay

December 23, 2020

Abstract

Containers are becoming an increasingly important part of the Linux ecosystem. Containerized package managers like Snapcraft [59] and FlatPak [18] enable easy distribution and dependency management for desktop applications, while others such as Docker [14], Kubernetes [26], and OpenShift [46] provide a framework for scaling and composing micro-services, especially in the cloud. While containers offer a convenient abstraction for distributing and configuring software, they are often used as light-weight alternatives to heavier virtualization techniques, such as virtual machines. Thus, containers can also be thought of as security mechanisms, providing some form of isolation between processes that share the underlying operating system's resources.

Despite this clear security use case, existing container implementations do not consider security a primary goal and often fall back to insecure defaults when the host does not support the correct security abstractions. Further, container security implementations are often complicated, relying on a myriad of virtualization techniques and security abstractions provided by the host operating system to isolate processes and enforce least-privilege. These security abstractions often paradoxically require elevated permissions to use in the first place, resulting in additional security risks when applications can escape confinement.

This paper presents BPFContain, a novel approach to containers under the Linux kernel designed to rectify these container security issues. BPFContain is built from the ground up as a light-weight yet secure process confinement solution for modern applications. Implemented in eBPF, an emerging technology for safely extending the Linux kernel, BPFContain enforces least-privilege in containerized applications without requiring any additional privileges from the host operating system. Users write policy in a high-level language designed to be readable and modifiable without requiring significant security expertise. To further secure the host-container boundary, BPFContain also enforces a secure implicit policy on all containers and provides additional kernel hardening features to prevent privilege escalation. In this paper, I describe BPFContain's design and implementation, evaluate its performance and security, and discuss how it compares with existing container solutions.

1 Introduction

Containers offer a light-weight alternative to traditional hypervisor-based virtualization techniques by leveraging a thin virtualization layer provided by the operating system and sharing the operating system kernel and resources [68, 69]. Thanks to their low overhead, high scalability, and composability, containers see increased industry adoption, particularly in multi-tenant platforms such as the cloud [68]. Despite these performance and convenience advantages, containers suffer from weaker security than traditional virtualization techniques, in part because they must share the underlying operating system kernel [32, 68, 69]. Another major factor impacting container security is the relatively lax attitude that many container management frameworks take toward least-privilege enforcement [68]. Popular container frameworks such as Docker [14] provision overly-permissive default access, rely on a complex and often ill-suited suite of security mechanisms provided by the host system, and support insecure configuration options [14, 17, 32, 68].

This paper proposes BPFContain, a novel approach to container security under the Linux kernel, to rectify overprivileged and insecure containers. Leveraging a new Linux kernel technology called eBPF, BPFContain uses runtime security instrumentation to implement container-specific policy enforcement and harden the host kernel against privilege escalation attacks mounted from containers. Specifically, BPFContain attaches eBPF programs to Linux's LSM (Linux Security Modules) hooks and critical functions within the kernel to enforce per-container policy in kernelspace. BPFContain combines user-defined policy files, implicit rules for container safety, and sensible defaults to enforce least-privilege access to sensitive system resources and secure the host-container boundary.

In summary, I offer the following contributions:

- I present BPFContain, a novel least-privilege enforcement mechanism for containers, leveraging eBPF programs attached to LSM hooks in the kernel. BPFContain improves container security by offering a container-specific LSM that can be dynamically loaded at runtime. Using eBPF, it also integrates with parts of the kernel that are not directly covered by LSM hooks, hardening the host against kernel privilege escalation attacks mounted from a container. Thanks to eBPF's dynamic instrumentation capabilities, this integration occurs at runtime and requires no modification or patching of the kernel.

- I propose a simple YAML-based policy language that offers optional layers of granularity to meet various use cases and multiple stakeholders' needs. The policy language is expressive enough that it can be used to confine individual system resources, yet simple enough that it affords ad-hoc confinement use cases.

- Using the BPFContain prototype as a proof of concept, I argue that it can be integrated directly with existing container management frameworks without modifying their source code, offering significant security advantages over traditional approaches to container least-privilege.

I present opportunities for future work that will allow BPFContain to both significantly streamline its policy language while simultaneously improving upon the status quo in confinement.

The rest of this paper proceeds as follows. Section 2 presents background on process confinement, containers, container security, and eBPF. Section 3 presents the container security threat model and discusses the motivation for implementing a new solution for container security. Section 4 examines the design and implementation of BPFContain, including its policy language and enforcement mechanisms. Section 5 presents the methodology that will be used to conduct a formal evaluation of BPFContain's usability, security, and performance. Section 6 discusses limitations and opportunities for future work. Section 7 covers related work and Section 8 concludes.

2 Background

2.1 The Process Confinement Problem

The *process confinement problem*, also known as the *sandboxing problem*, refers to the goal of isolating a process or group of processes from the rest of the running system. In practice, this is often achieved by restricting an application's possible behaviour to its desired functionality, explicitly targeting its access to security-sensitive system resources such as files, network interfaces, and other running applications. Despite decades of work following Lampson's [28] first proposal of the process confinement problem in 1973, process confinement remains a somewhat open problem to date [12].

Anderson's 1972 introduction of the reference monitor architecture [1] has informed the design and implementation of operating system access control mechanisms for nearly five decades. As subjects request access to system resources, the kernel's reference monitor validates requests and makes policy decisions. While all commodity operating systems support at least a basic notion of access control, many implement a coarse-grained, discretionary access control mechanism that is insufficient for providing the fine-grained protection required in security-sensitive or multi-tenant environments. For additional protection, we must turn to more advanced security features offered by the operating system.

A separate yet related problem involves constructing policy languages with which to express security policy. These policy languages need to offer a careful balance of terseness and expressiveness. A policy language must be terse enough to enable human policy authorship and audit yet expressive enough to enforce least-privilege. Similarly, careful consideration is required when developing policy language abstractions. Close conformity with the system's underlying semantics may improve policy expressiveness but impact usability. Significant abstractions might similarly impact the expressiveness of the policy language.

These issues of policy enforcement and policy language design are at the heart of the process confinement problem. This paper seeks to tackle this problem from the perspective of containers

and container management.

2.2 Low-Level Isolation Techniques

The Linux kernel supports various lower-level abstractions for implementing virtualization and enforcing least-privilege. While many of these mechanisms are insufficient for a full confinement implementation on their own, they are typically used, in *combination*, by higher-level techniques such as containers (c.f. Section 2.3) to achieve confinement. This section covers these low-level abstractions in detail.

Unix Discretionary Access Control Discretionary access control (DAC) forms the most basic access control mechanism in many operating systems, including popular commodity operating systems such as Linux, macOS, and Windows. First formalized in the 1983 Department of Defense standard [70], a DAC system partitions system objects (e.g. files) by their respective owners and allows resource owners to grant access to other users at their discretion. Typically, systems implementing discretionary access control also provide an exceptional user or role with the power to override discretionary access controls, such as the superuser (i.e. `root`) in Unix-like operating systems and the Administrator role in Windows.

While discretionary access controls themselves are insufficient to implement proper process confinement, they form the basis for the bare minimum level of protection available on many operating systems; therefore, they are an essential part of the process confinement discussion. In many cases, user-centric discretionary access controls are abused to create per-application "users" and "groups". For instance, a typical pattern in Unix-like systems such as Linux, macOS, FreeBSD, and OpenBSD is to have specific users reserved for security-sensitive applications such as network-facing daemons. The Android mobile operating system takes this one step further, instead assigning an application- or developer-specific UID (user ID) and GID (group ID) to *each* application installed on the device [21].

In theory, these abuses of the DAC model would help mitigate the potential damage that a compromised application can do to the resources that belong to other users and applications on the system. However, due to DAC's discretionary nature, nothing prevents a given user from granting permissions to all other users on the system, barring the use of other security measures. Further, non-human users' inclusion into a user-centric permission model may result in a disparity between an end-user's expectations and the reality of what a "user" is. This gap in understanding could result in further usability and security concerns.

POSIX Capabilities Related to discretionary access control are POSIX capabilities [8, 9, 34], which can be used to grant additional privileges to specific processes, overriding existing discretionary permissions. Further, a privileged process may *drop* specific capabilities that it no longer needs, retaining those it needs. Consequently, POSIX capabilities provide a finer-grained alternative

to the all-or-nothing superuser privileges required by certain applications. For instance, a web-facing process that requires access to privileged ports has no business overriding file permissions. POSIX capabilities provide an interface for making such distinctions. Despite these benefits, POSIX capabilities have been criticized for adding additional complexity to an increasingly complex Linux permission model [8, 9]. Further, POSIX capabilities do nothing to confine processes beyond the original DAC model. Instead, they help to solve the problem of overprivileged processes by limiting the privileges that they require in the first place.

Namespaces and Cgroups In Linux, *namespaces* and *cgroups* (short for control groups) allow for further confinement of processes by restricting the system resources that a process or group of processes can access. Namespaces isolate access by providing a process group a private, virtualized naming of a class of resources, such as process IDs, filesystem mountpoints, and user IDs. As of version 5.6, Linux supports eight distinct namespaces, depicted in Table 2.1. Complementary to namespaces, cgroups limit available *quantities* of system resources, such as CPU, memory, and block device I/O. Namespaces and cgroups provide fine granularity for limiting a process's view of available system resources. In this sense, they are better classified as a mechanism for implementing virtualization rather than least-privilege. They thus must be combined with other measures to constitute a full confinement implementation.

Table 2.1: Linux namespaces (as of kernel version 5.6) and what they can be used to isolate.

Namespace	Isolates
PID	Process IDs (PIDs)
Mount	Filesystem mountpoints
Network	Networking stack
UTS	Host and domain names
IPC	Inter-process communication mechanisms
User	User IDs (UIDs) and group IDs (GIDs)
Time	System time
Cgroup	Visibility of cgroup membership

System Call Interposition System call interposition has historically been a prevalent process confinement technique. Several frameworks exist today for system call interposition on various Unix-like operating systems [2, 44, 45, 73]. Since system calls are used to request services from the operating system, they define the interface with the operating system's reference monitor [1]. This property makes system call interposition a particularly attractive technique for the implementation of fine-grained policy enforcement mechanisms.

One of the earliest forays into system call interposition for process confinement was TRON [5]. Implemented in 1995 for the Unix operating system, TRON provided a kernelspace mechanism for enforcing *protection domains* on userspace processes. A TRON protection domain consists of its confined processes, a set of allowed operations, and a *violation handler*, which TRON invokes on policy violations. Processes configure protection domains and then employ a special `tron_fork` system call to spawn a confined child process. While TRON is not application transparent by itself, it does come with a set of userspace tools to abstract away the configuration of protection domains. Unfortunately, even with these higher-level userspace tools, TRON still assumes a certain degree of security expertise for a user to confine their applications properly.

Perhaps the most pervasive framework for interposing on system calls is ptrace [45], a process tracing and debugging framework that comes enabled in some form or another on all Unix-like operating systems. While ptrace itself is *not* designed process confinement, some research prototypes [20, 71] have leveraged it in the past. Unfortunately, ptrace is not generally considered production-safe due to its high overhead and buggy interactions with more complex programs such as sendmail, which is especially problematic considering that these are the types of programs that we often wish to confine.

Janus [20, 71] was an early exploration of process confinement using Solaris' version of ptrace. In Solaris, ptrace provides a library call interface into the procfs virtual file system and allows tracer applications to make filtering decisions on behalf of traced processes while interposing on system calls. Janus was later ported to Linux using a modified version of Linux's `ptrace(2)` system call [71]. In Janus, a supervisor process reads a policy file and attaches itself to a confined process with ptrace. From there, security-sensitive system calls in the confined application are forwarded to the Janus supervisor process to make a policy decision. However, this approach adds considerable overhead to confined processes because ptrace requires *multiple* context switches between userspace and kernelspace to coordinate between the tracer and tracee.

To implement its policy language, Janus defines higher-level interfaces into various groups of system calls, called *policy modules*. These policy modules can filter groups of related system calls by parameterizing them with allowed actions and system objects. While this abstraction is helpful to group system calls by their related functionality, it does little to help Janus' usability, which is still tightly coupled with the underlying system calls. This lack of usability makes it difficult for a non-expert user to write an effective Janus policy.

Anderson published a study in the FreeBSD journal [2] comparing three system call interposition frameworks for three distinct Unix-like operating systems: Linux's seccomp-bpf [15, 36], OpenBSD's pledge [44], and FreeBSD's Capsicum [72, 73]. While these three frameworks all interpose on system calls, they do so with varying degrees of security, complexity, and granularity [2], so each merits study in its own regard.

In the original Linux seccomp implementation, processes use a special `seccomp(2)` system call to enter a secure computing state. By default, processes that have entered this state are restricted

to performing `read(2)`, `write(2)`, `sigreturn(2)`, and `exit(2)` system calls. Pragmatically, this means that a process could read and write on its open file descriptors, return from invoked signal handlers, and terminate itself. All violations of this policy would result in forced termination. In a 2012 RFC [15], Drewry introduced an extension to seccomp, enabling the use of BPF programs for the defining filters on system call arguments. This extension, dubbed seccomp-bpf, enables creating fine-grained seccomp policies that filter on system call numbers and arguments, providing a high degree of control to applications that wish to sandbox themselves.

Despite the high degree of control that seccomp-bpf offers to applications, it has severe usability and security concerns, rendering it an unacceptable solution for ad-hoc confinement by end-users. Classic BPF [41] is a rather arcane bytecode language, and writing Classic BPF programs by hand is a task left only to expert users. Further, seccomp-bpf policy is easy to misconfigure, resulting in potential security violations; for instance, an attacker may entirely circumvent a policy that specifies restrictions on the `open(2)` system call but not `openat(2)`. Finally, despite userspace library efforts to abstract away the underlying BPF programs [52], seccomp-bpf remains accessible only to application developers with significant security expertise.

OpenBSD's pledge [44] takes a more straightforward, coarser-grained approach to system call filtering than seccomp-bpf, instead grouping system calls into high-level semantically meaningful categories, such as `stdio` which includes `read(2)` and `write(2)`, for example [2]. This coarse granularity and simplicity provide increased usability but come at the expense of expressiveness. There is no canonical way to distinguish subsets of system call groups or filter system calls by their arguments. Despite its increased usability for developers, pledge still suffers from a lack of application transparency just as seccomp-bpf does, meaning that it is only suitable for application developers rather than end-users.

Unlike seccomp-bpf and pledge, which apply filtering rules to system calls directly, FreeBSD's Capsicum takes the approach of restricting access to global namespaces via a capability-based implementation [73]. In Capsicum, a process enters *capability mode* using a special `cap_enter` system call. Once in capability mode, access to global namespaces is restricted to the capabilities requested by the process. These capabilities are inherited across `fork(2)` and `execve(2)` calls. Much like seccomp-bpf and pledge, however, Capsicum is *not* application transparent and is designed for use by developers rather than end-users.

Linux Security Modules The Linux Security Modules (LSM) API [74] provides an extensible security framework for the Linux kernel, allowing for the implementation of powerful kernelspace security mechanisms that can be chained together. LSM works by integrating a series of strategically placed *security hooks* into kernelspace code. These hooks roughly correspond with boundaries for the modification of kernel objects. Multiple security implementations can hook into these LSM hooks and provide callbacks that generate audit logs and make policy decisions. Figure 2.1 depicts the LSM architecture in detail.

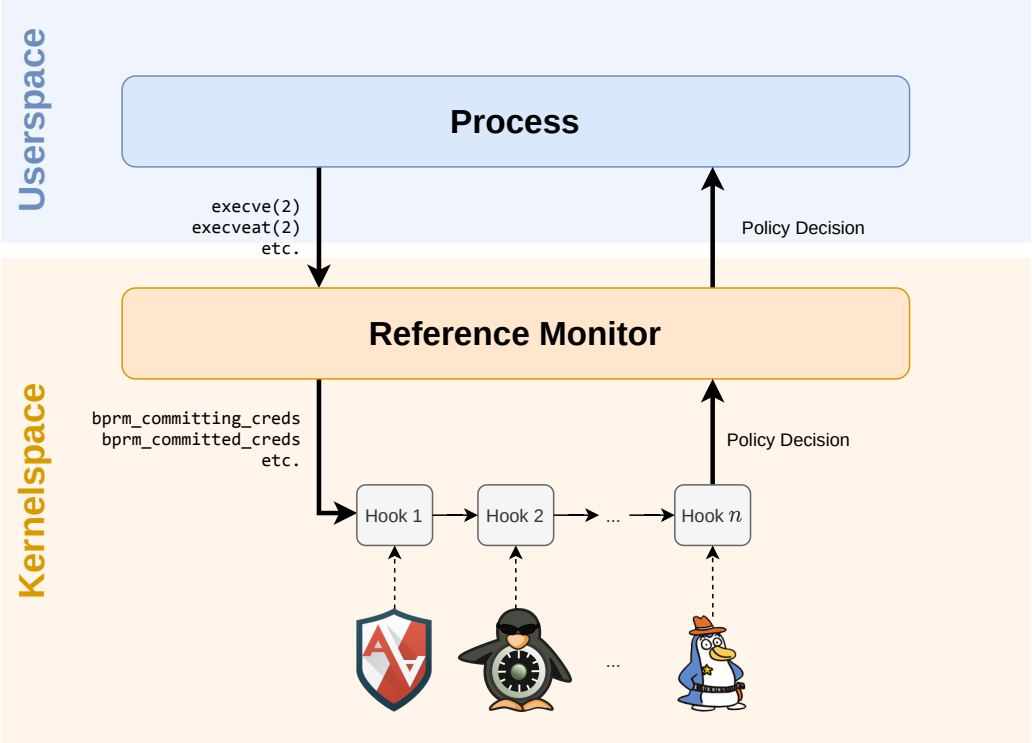

Figure 2.1: The LSM architecture. Note the many-to-many relation between access requests and hook invocations. Multiple LSM hooks may be chained together, incorporating policy from many security mechanisms. All hooks must agree to allow the access or it will be denied.

The LSM API sits at a level of abstraction just above the system call API—a single LSM hook may cover multiple system calls, and a single system call may contain multiple such LSM hooks. For instance, the `execve(2)` and `execveat(2)` calls both result in a call to the `bprm_committing_creds` and `bprm_committed_creds` hooks (among others). This design provides a excellent level of abstraction compared to system-call-based approaches like seccomp-bpf [15, 36] in that a single LSM hook can cover all closely related security events (recall the issue of `open(2)` vs. `openat(2)` in seccomp-bpf).

The Linux kernel contains several in-tree LSM-based security modules, which may be enabled by default on certain distributions. Many such modules implement *mandatory access control* (MAC) schemes, which enable fine-grained access control that can limit *all users' privileges*—even the superuser. SELinux [57] and AppArmor [11] are two such MAC LSMs, each with distinct policy semantics. I discuss each in turn.

SELinux [57] was initially developed by the NSA as a Linux implementation of the Flask [61] security model. Under SELinux, system subjects (e.g. users or processes) and system objects (e.g. files or network sockets) are assigned corresponding labels. Security policy is then written based on these labels, specifying the allowed access patterns between a particular object type and subject type. SELinux's policy language is famously arcane [51]. Despite multiple efforts to introduce automated policy generation [40, 58, 60], writing and auditing SELinux security policy remains a task for security experts rather than end-users. Further, due to the difficulty of writing and auditing the complex

SELinux policy language, there is a natural tendency for human policy authors to err on the side of over-permission, violating the principle of least privilege.

AppArmor (originally called SubDomain) [11] is often touted as a more usable alternative to SELinux, although usability studies have shown that this claim merits scrutiny [51]. Rather than basing security policy on labelling system subjects and objects, AppArmor instead employs path-based enforcement. AppArmor defines policy in per-application profiles, which contain rules specifying what system objects the application can access. System objects are identified directly (for example, via pathnames, socket classes, or IP network addresses) rather than labelled. AppArmor also supports the notion of *changing hats*, through which a process may change its AppArmor profile under certain conditions specified in the policy. Although AppArmor profiles are more conforming to standard Unix semantics than their SELinux counterparts, users who wish to write AppArmor policy still require a considerable amount of knowledge about operating system security [51].

2.3 Containers

Containers use OS-level virtualization and confinement mechanisms (c.f. Section 2.2) to provide a (semi-)isolated environment for the execution of processes [68]. Since they run directly on the host operating system and share the underlying OS kernel, containers do not require a full guest operating system to implement virtualization. This technique has the advantage of offering a light-weight alternative to traditional hardware virtualization approaches using full virtual machines [68]. Compared with containers, traditional approaches to virtualization involve hypervisors, which virtualize and provide access to the underlying hardware, either running on top of a host operating system or directly on top of the hardware itself. Full virtual machines run on top of these hypervisors, each running a guest operating system with a full userland and kernel. Full virtualization provides stronger isolation guarantees than containers but involves significantly more overhead imposed by the guest operating system [68]. Figure 2.2 depicts an overview of the architectural differences between containers and full hardware virtualization solutions (i.e. virtual machines running on top of hypervisors).

Since containers must share the host operating system kernel and related resources, it is essential to consider how best to isolate them from one another. Therefore, a container management system generally seeks to achieve the following security-related goals[1]:

CG1. Virtualization. Virtualization aims to provide each container with a *virtual view* of system resources. Containers generally achieve virtualization using a combination of Linux namespaces, cgroups, and filesystem mounts. Namespaces provide a private view of enumerable resources (i.e. a virtual mapping of IDs to resources). Such enumerable resources include process IDs, user and group IDs, mountpoints, and the network interfaces. Cgroups

[1]Dependency management is another goal of container management systems like Docker [14], Kubernetes [26], and OpenShift [46], but it is out of scope for his paper.

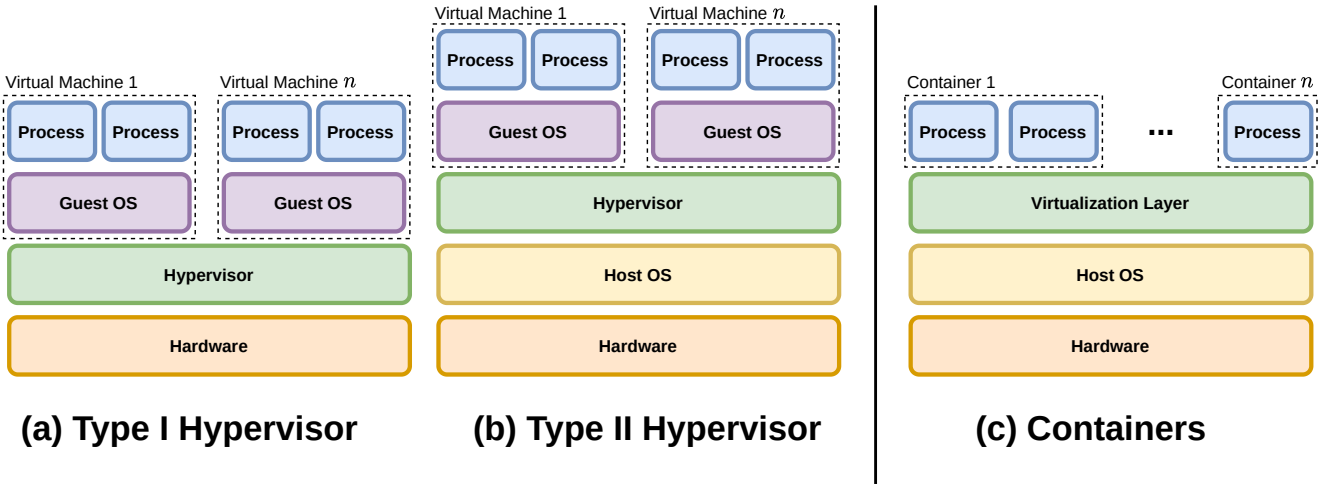

Figure 2.2: Virtual machine and container architectures. Type I hypervisors **(a)** virtualize and control the underlying hardware directly, but require full guest operating systems on top of the virtualization layer. Type II hypervisors **(b)** run on top of a host operating system but still require full guest operating systems above the virtualization layer. Containers **(c)** achieve virtualization using a thin layer provided by the host operating system itself. They share the underlying operating system kernel and resources, requiring no guest operating system [68].

similarly provide a virtual view of *quantifiable resources* (i.e. how much of a given resource is available). Such quantifiable resources include the CPU, persistent storage, memory, and I/O bandwidth. Filesystem mounts, combined with mount namespaces, provide a virtual view of visible files.

Although full virtualization may be desirable in the ideal case, containers often only implement partial virtualization [32, 68] due to various factors. Pragmatically, it is often beneficial for containers to have a shared view of specific system resources, depending on the use case. For instance, two containers might share a copy of the same shared library or require access to a shared IPC namespace to enable communication. In practice, containers often leverage layered filesystems such as overlayfs [16] to deduplicate files across containers and the host system. Partial virtualization can enable lighter-weight containers and easier communication between two containers to satisfy composability [68].

CG2. Least-Privilege. For a container to be considered secure, it must enforce least-privilege on its processes [68]. This requirement makes practical sense, given that a container runs directly on the host system and must share the underlying OS kernel and resources with both other containers and the host system itself. Without least-privilege, a process running in a container has virtually the same access rights as an unconfined process. When the container itself is running with privileged access to the system (as is often the case [32, 68]), this may even result in an *escalation of privilege* compared to the scenario where the process runs directly on the host. For these reasons, it is neither practical nor advisable to rely on weak virtualization guarantees to protect the host system with no means of

enforcing least-privilege [68].

A least-privilege implementation for containers typically involves a combination of multiple enforcement mechanisms, including Unix DAC, seccomp-based system call filtering, dropped POSIX capabilities, and mandatory access control mechanisms (using one of the Linux MAC LSMs) [14, 32, 68]. This complexity can lead to usability and auditability concerns, as a simple policy language must compile down to multiple complex enforcement mechanisms that need to work cooperatively [17].

Despite its evident importance for container security, existing container management solutions generally treat least-privilege as a secondary goal [68]. Docker attempts to provide sensible security defaults for containers. Still, these defaults may be easily overridden and often rely on extra kernel security features such as the AppArmor LSM [14]. When AppArmor is not available, Docker falls back to relying exclusively on its default seccomp policy and dropped capabilities. Security defaults for containers also often do not adhere to the principle of least privilege. For instance, Docker provides containers with 15 Linux capabilities by default, including `CAP_DAC_OVERRIDE`, which allows a container to override all discretionary access control checks [14, 68].

CG3. Composability. Increasingly, containers are being used to implement composable microservices [68]. For instance, Kubernetes [26] allows the user to group containers into *pods*, allowing them to communicate with each other in pre-defined ways. For composability, a container needs the ability to communicate with another container without sacrificing virtualization or least-privilege. In practice, containers achieve such composability by defining specific inter-container exceptions to virtualization and least-privilege policy [68]. Naturally, these exceptions can increase the risk of an insecure configuration, and the user must carefully manage them to avoid overprivilege.

2.4 Classic and Extended BPF

The original Berkeley Packet Filter (BPF) [41], hereafter referred to as Classic BPF, was a packet filtering mechanism implemented initially for BSD Unix. McCanne and Jacobson created Classic BPF as a light-weight replacement for traditional packet filtering mechanisms, which relied on frequent context switches between userspace and kernelspace while making filtering decisions. Instead, Classic BPF implemented a simple register-based virtual machine language and efficient buffer data structures to minimize the required context switches. As an efficient packet filtering mechanism, Classic BPF quickly gained traction in the *NIX community and was subsequently ported to various open-source Unix and Unix-like operating systems, most notably Linux [33], OpenBSD [43], and FreeBSD [19].

The Linux kernel development community eventually realized that the BPF engine could be applied to more than just packet filtering. The 2012 introduction of seccomp-bpf [15, 36] enabled

Classic BPF programs to be written and applied to make system call filtering decisions for userspace applications. This extension to seccomp transformed it into a powerful (yet notoriously difficult-to-use [2]) mechanism for making security decisions about system calls in a confined process.

In 2014, Starovoitov and Borkmann merged a complete rewrite of the Linux BPF engine, dubbed Extended BPF (eBPF), into the mainline kernel [66]. eBPF expands on the original BPF specification by introducing:

- An extended instruction set;
- 11 registers (10 of which are general-purpose);
- Access to allow-listed kernel helpers;
- Just-in-time (JIT) compilation to native instruction sets;
- A program safety verifier;
- A large collection of specialized data structures; and
- New program types which can be attached to a variety of system events in both userspace and kernelspace.

These extensions to the Classic BPF engine effectively turn eBPF into a general-purpose execution engine in the kernel with powerful system introspection and kernel extension capabilities. eBPF programs execute in the kernel with supervisor privileges but are limited by a restricted execution context and pre-checked for safety by an in-kernel verification engine. In particular, eBPF programs are limited to a 512-byte stack, cannot access unbounded memory regions, must not have back-edges in their control flow, and must provably terminate [22]. As a consequence of these restrictions, eBPF programs are not Turing-complete. Where necessary, an eBPF program can make calls to a set of allow-listed kernel helpers to obtain additional functionality, such as access to external memory regions and various kernel facilities such as signalling or random number generation [22].

A privileged userspace process may load an eBPF program into the kernel using Linux's `bpf(2)` system call. While it is possible to write eBPF bytecode by hand [22], several front-ends exist for compiling eBPF bytecode from a restricted subset of the C programming language[2], including bcc [23] and libbpf [31]. These front-ends typically use the LLVM [38] compiler toolchain to produce BPF bytecode. When the kernel receives a request to load an eBPF program, it first checks the bytecode to ensure that it conforms to the safety invariants outlined above. If the verifier accepts the program, it may then be attached to one or more system events. When an event fires, the eBPF program is executed via just-in-time compilation to the native instruction set. eBPF programs can store data in several specialized in-kernel data structures, made accessible to userspace via the `bpf(2)` system call or a direct memory mapping. Figure 2.3 depicts this process in detail.

[2]In principle, this language need not be C. For instance, a framework exists for writing eBPF programs in pure Rust [47]. However, C is a popular choice since it is tightly coupled with the underlying implementation of the kernel.

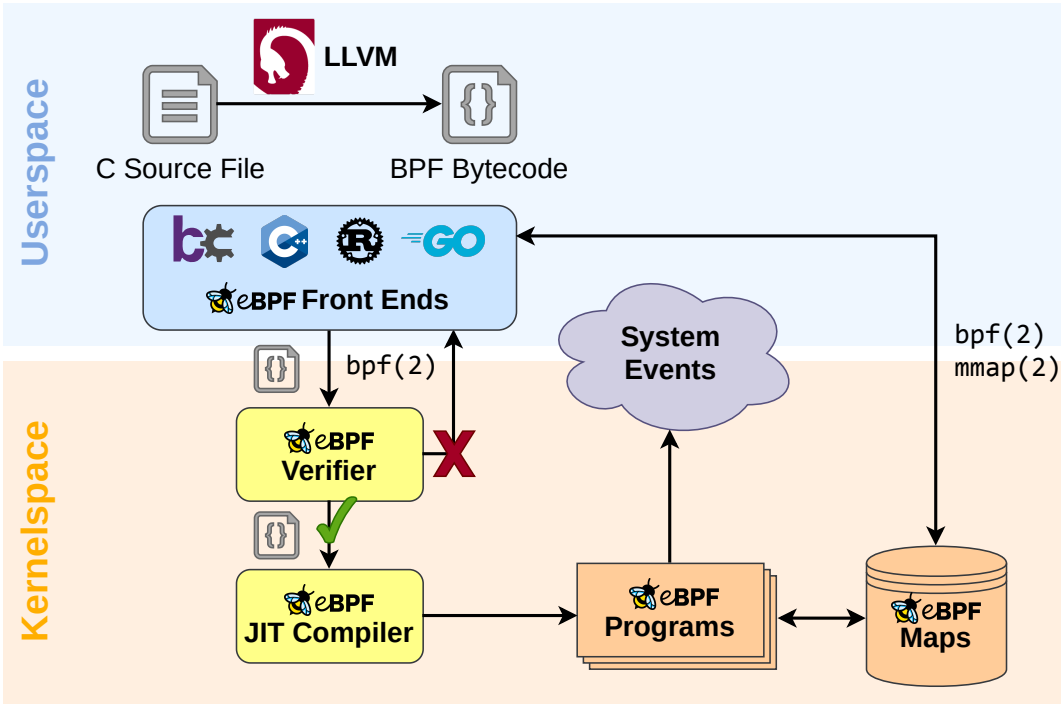

Figure 2.3: eBPF mechanisms in the kernel. Userspace front-ends compile C source code into eBPF bytecode using the LLVM toolchain and load it into the kernel with the `bpf(2)` system call. The in-kernel verifier either accepts or rejects the program based on its adherence to safety invariants. Accepted programs are attached to system events across the userspace and kernelspace boundary where they are just-in-time compiled to the native instruction set. Programs can store and fetch data using data structures called "eBPF maps", which can also be accessed directly from userspace.

3 Motivation

3.1 Threat Model

Sultan *et al.* [68] propose four broad categories of container-related threats. Figure 3.1 presents an overview of each category. An application running within a container might attack the container itself, attempting to escape confinement or interfere with the execution of co-located applications running within the same container. Inter-container threats are similarly possible, wherein one container attempts to interfere with or take over another. Since containers share the underlying host operating system, it is also possible for a container to directly attack the host, either by escaping confinement altogether or by launching denial of service or resource consumption attacks. Finally, a malicious or semi-honest host system may attack containers running within it. Researchers have generally recognized that mitigating this fourth category of attack requires the use of hardware security mechanisms [7, 32, 68] such as trusted execution environments or trusted platform modules. Such host-to-container attacks are, therefore, out of scope for this paper.

In this threat model, we consider three broad classes of attack vector, comprised of the containers and the applications that run within them. These attack vectors are described in turn below.

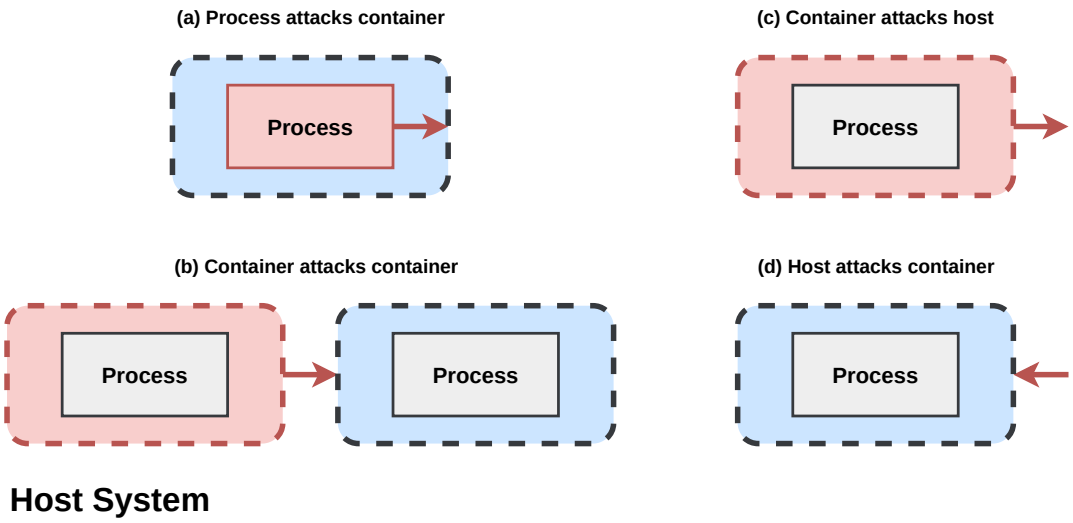

Figure 3.1: Four categories of container-related attacks [68]. **(a)** A process running within a container attacks the container itself. **(b)** One container attacks another container. **(c)** A container attacks the host system. **(d)** The host system attacks the container. This fourth category of attack is out of scope for this paper.

AV1. Malicious Applications. A malicious application is designed with express malicious intent. Malicious software may actively attempt to subvert other applications, other containers, or the host system itself. This subversion could include privilege escalation attacks on the host system, denial of service attacks on other containers or the host, or the installation of backdoors. A sophisticated attacker could even abuse a malicious application to install a rootkit [3] in the container, on the host system, or in the host's firmware, stealthily gaining permanent and possibly overprivileged access.

AV2. Semi-Honest Applications. In contrast with malicious applications, semi-honest applications are not necessarily expressly designed with malicious intent and might even be cooperative with other applications or containers. However, the semi-honest application can passively participate in unwanted activity, such as surveillance or consumption of the host's resources.

AV3. Vulnerable Applications. Vulnerable applications running inside containers may become compromised by attackers, often with the goal of using these applications to orchestrate a more sophisticated attack on other applications running within the same container, other containers, or the host system itself [68]. Common vulnerabilities here include code execution vulnerabilities on untrusted input, memory corruption bugs, and privilege escalation vulnerabilities in a container's configuration. Exploitation of kernel-level vulnerabilities are also a concern here, as an attacker can potentially abuse a legitimate application to target vulnerable code paths in the kernel [32].

An attacker could have several goals under this threat model. Such goals might include [32, 68]:

AG1. Escalation of Privilege. An attacker that manages to escalate privileges within the context of a container may escape confinement altogether and interfere with the host or with other containers. In the worst case, the attacker may obtain root privileges and establish total control over the host system.

AG2. Denial of Service. An attacker might abuse a poorly configured container to mount a denial of service attack on the host system or other containers. For instance, an attacker could consume resources on the host system, disable network interfaces, unmount filesystems, or kill other running processes.

AG3. Remote Code Execution. Remote code execution vulnerabilities could allow an attacker to control a container or, in the worst case, the host itself. Kernelspace code execution vulnerabilities are particularly dangerous, as they can often be used for escalation of privilege, information leakage, or denial of service and affect the entire host system [32, 68].

AG4. Information Disclosure. An attacker might disclose sensitive information, such as API keys, secret cryptographic keys, passwords, or other sensitive user data. In the worst case, they might be able to disclose information belonging to another container or the host itself.

AG5. Tampering. An attacker could tamper with other processes running within a container, other containers, or the host itself. Such tampering attacks might include replacing or modifying file systems or data to deliberately induce incorrect computational results [68].

AG6. Backdoor Establishment. An attacker can establish a backdoor to obtain permanent or semi-permanent access to a container or the host system itself. For instance, an SSH backdoor (e.g. installing an attacker-controlled public key into SSH's list of authorized keys) can enable persistent network intrusions on the host system. In the worst case, this might involve installing a rootkit [3] in the host operating system to enable stealthy, overprivileged access.

Since containers are often co-located in large-scale, multi-tenant systems such as the cloud [68], the potential for exploitation or abuse by foreign threat actors is exacerbated. Thus, container management systems must enforce least-privilege on containers to prevent such exploitation from negatively impacting the rest of the system. A secure container management system with sensible defaults and strong protection mechanisms would defeat most, if not all, of the attacks outlined above.

3.2 The Quest for Secure Containers

Sultan *et al.* cite container security issues as the primary factor inhibiting their widespread adoption [68]. In existing container management systems, least-privilege is often treated as a secondary goal. Instead, light-weight virtualization, dependency management, convenience, and environmental reproducibility are prioritized. For instance, Docker [14, 68] provides an overly permissive default security policy, enabling the use of several dangerous system calls and POSIX capabilities. Extra protection, such as the system call restrictions offered by its default seccomp and AppArmor policies, is considered opt-in rather than opt-out [14, 68]. Further, in an environment where running AppArmor is not an option, Docker must forgo the protection offered by its AppArmor policy altogether.

Other container security issues arise when attackers can bypass protection mechanisms altogether. For instance, code execution attacks in kernelspace can be mounted by exploiting vulnerabilities in the kernel's API and bypassing memory protection mechanisms. Depending on the attack, the interfaces required to mount such an exploit successfully may not even be gated by a container security mechanism. Such exploits effectively reduce security to that of a process running directly on the host system. While these attacks may be difficult to mount in practice, they still merit consideration.

Unlike full hardware virtualization under a hypervisor, containers must share the underlying host operating system kernel and resources. In practice, this means that their attack surface is fundamentally larger than that of a virtual machine. If we are to strive for genuinely secure containers, we must seek to minimize this attack surface as much as possible. This means that we need to build containers to be secure from the ground-up, prioritizing least-privilege over all other goals.

4 BPFContain Design and Implementation

4.1 Design Goals

Five specific goals informed the design of BPFContain's policy language and enforcement mechanism, enumerated below as Design Goals D1 to D5.

D1. Usability. BPFContain should not impose unnecessary usability barriers on end-users. Its policy language should be easy to understand and semantically meaningful to users without requiring significant security knowledge. To accomplish this goal, BPFContain takes some inspiration from other high-level policy languages for containerized applications, such as those used in Snapcraft [59].

D2. Configurability. It should be easy for an end-user to reconfigure policy to match their specific use case without worrying about the operating system's underlying details or the

policy enforcement mechanism. It should be possible to use BPFContain to restrict specific unwanted behaviour in a given application without writing a rigorous security policy from scratch.

D3. Transparency. Confining an application using BPFContain should not require modifying the application's source code or running the application using a privileged SUID (Set User ID root) binary. BPFContain should be entirely agnostic to the rest of the system and should not interfere with its regular use.

D4. Adoptability. BPFContain should be adoptable across various system configurations and should not negatively impact the running system. It should be possible to deploy BPFContain in a production environment without impacting system stability and robustness or exposing the system to new security vulnerabilities. BPFContain relies on the underlying properties of its eBPF implementation to achieve its adoptability guarantees.

D5. Security. BPFContain should be built from the ground up with security in mind. In particular, security should not be an opt-in feature, and BPFContain should adhere to the principle of least privilege [50] by default. It should be easy to tune a BPFContain policy to respond to new threats.

4.2 Architectural Overview

BPFContain consists of both userspace and kernelspace components, which interact co-operatively to implement its policy enforcement mechanism. Roughly, its architecture (depicted in Figure 4.1) can be broken down into the following four components:

C1. A privileged daemon, responsible for loading and managing the lifecycle of eBPF programs and maps and logging security events to userspace.

C2. A small shared library and unprivileged wrapper application used to initiate confinement.

C3. A set of eBPF programs, running in kernelspace. These programs are attached to LSM hooks in the kernel as well as the shared library in C2.

C4. A set of eBPF maps, special data structures which allow bidirectional communication between userspace and kernelspace. These maps are used to track the state of running containers and store the active security policy for each container.

In userspace, BPFContain is implemented as a privileged daemon based on the bcc [23] eBPF framework for Python. The daemon is responsible for loading BPFContain's eBPF programs and maps and logging security events to userspace, such as policy violations. When it first starts, the daemon invokes a series of `bpf(2)` system calls to load its eBPF programs and maps into the kernel.

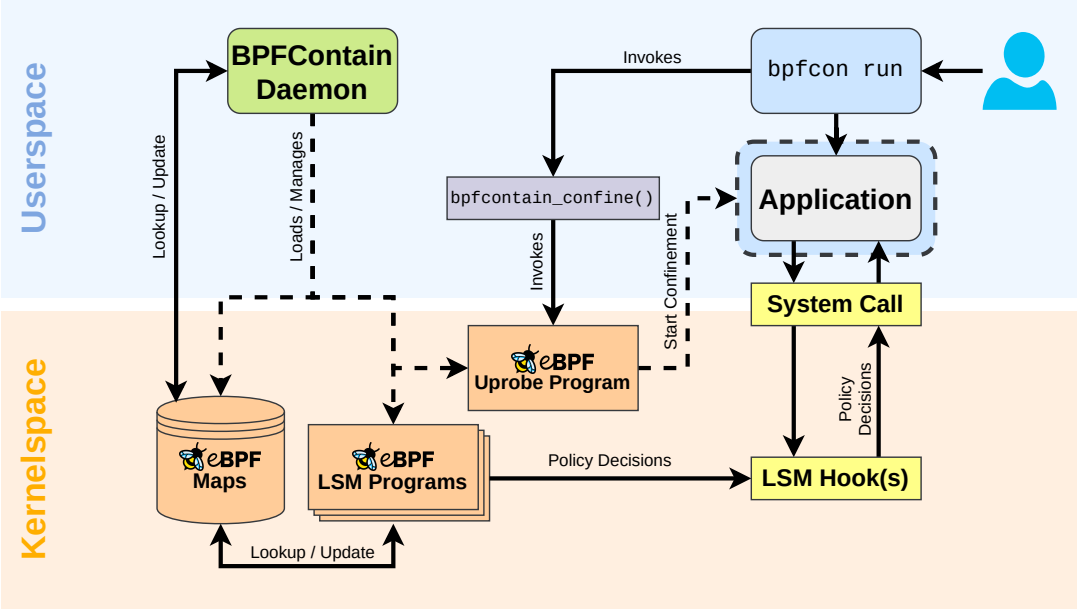

Figure 4.1: A diagram of BPFContain's architecture. The privileged daemon (green) is responsible for loading the necessary eBPF maps and programs (orange) into the kernel and managing their lifecycle. The user starts a container by executing an unprivileged wrapper application (blue), which invokes the `bpfcontain_confine` library call (purple), trapping to a special eBPF program that associates the process group with the right policy. When the confined application (grey) makes a system call to request access to a sensitive resource, the kernel invokes one or more LSM hooks (yellow), which trap to corresponding eBPF LSM programs that make the correct policy decision.

After loading all eBPF programs and maps, the daemon then parses, translates, and loads each per-container policy file into several eBPF maps.

For the bulk of its policy enforcement, BPFContain leverages the KRSI (kernel runtime security instrumentation) patch merged by KP Singh [10, 54] into the Linux 5.7 kernel. This patch enables eBPF programs to be attached to LSM hooks at runtime, making policy decisions and log security events. These programs work cooperatively with each other and with any other LSMs running on the system to come to a policy decision. When an event fires, the LSM programs query policy from policy maps to come to a decision. This dynamic runtime policy enforcement is at the core of BPFContain's flexible container security approach.

4.3 BPFContain Policy

BPFContain policy consists of simple manifests written in YAML [4], a human-readable data serialization language based on key-value pairs. Each BPFContain container is associated with a manifest, consisting of a few lines of metadata followed by a set of *rights* and *restrictions*. A *right* specifies access that should be granted to a container, while a *restriction* is used to revoke access. BPFContain compartmentalizes rights and restrictions into various granularities. For instance, a file right has a finer granularity than a filesystem right. A restriction always overrides a right,

except in the case where the right has finer granularity. As an example, a right to access the root filesystem would be overridden by a restriction on the user's home directory. In practice, this allows the construction of nuanced policies that specify coarse-grained access with finer-grained exceptions. Table 4.1 describes the various access labels that can be used in BPFContain policy.

Table 4.1: Access labels for BPFContain policy along with their parameters and descriptions. Square brackets denote an optional parameter.

Resource	Parameters	Description
`filesystem`	Mountpoint, [Access]	Grants or revokes access at the granularity of a filesystem mountpoint. Access defaults to read, write, append, chdir, create, and setattr, unless otherwise specified. An optional `readonly` flag grants read and chdir access. An optional `appendonly` flag grants read, append, setattr, create, and chdir access.
`file`	Pathname, Access	Grants or revokes access at the granularity of individual files. Access must be specified as a string of access flags (see Table 4.2).
`net-bind`		Grants or revokes access to the `CAP_NET_BIND_SERVICE` POSIX capability, allowing the container to bind to privileged ports.
`net-raw`		Grants or revokes access to the `CAP_NET_RAW` POSIX capability, allowing the container to use raw sockets.
`net-broadcast`		Grants or revokes access to the `CAP_NET_BROADCAST` POSIX capability, allowing the container to broadcast and listen to multicast network traffic.
`dac-override`		Grants or revokes access to the `CAP_DAC_OVERRIDE` POSIX capability, allowing the container to override discretionary access controls.
`network`	[Family], [Access]	Grants or revokes access to network communications. A specific address family and access pattern may optionally be specified.
`ipc`	Container	Grants or revokes access to communicate with processes in *another* BPFContain container. This covers all supported System V IPC categories as well as Unix sockets and signals. Both containers must mutually declare IPC access.
`tty`	[Access]	Grants or revokes access to terminal devices.
`pts`	[Access]	Grants or revokes access to pseudo-terminal devices.
`video`	[Access]	Grants or revokes access to video devices.
`sound`	[Access]	Grants or revokes access to sound devices.

`graphics`	[Access]	Grants or revokes access to graphics devices.
`random`	[Access]	Grants or revokes access to random and urandom devices.
	...	

BPFContain provides three policy granularities for file access: filesystem rules, file rules, and device rules. A filesystem rule grants access to an entire filesystem, specified by providing the pathname of its mountpoint. For instance, a policy would specify access to the root filesystem with `filesystem /` and procfs with `filesystem /proc`. File rules specify access at the granularity of individual files and directories. Finally, coarse-grained device rules such as `tty`, `pts`, `video`, and `sound` specify access to common character and block devices. Each file access rule supports 13 specific access categories (outlined in Table 4.2), which may be combined as necessary. Filesystem rules also support two coarse-grained access flags, `readonly` and `appendonly`, which act as shortcuts for commonly-used access flags for filesystems.

Table 4.2: Access categories for filesystem and file policy.

Access	Flag	Description
Read	`r`	The container may read the file. This does not apply to directories.
Write	`w`	The container may write to the file. This does not apply to directories.
Execute	`x`	The container may execute the file (via execve). This does not apply to directories.
Append	`a`	The container may append to the file (without overwriting existing contents). This does not apply to directories.
Create	`c`	The container may create new files in the directory.
Rename	`n`	The container may rename the file or directory.
Delete	`d`	The container may delete the file or directory.
Change Directory	`t`	The container may change directory into this directory.
Set Attribute	`s`	The container may set attributes on this file or directory.
Change Permissions	`p`	The container may change permissions on the file or directory.
Change Owner	`o`	The container may change the owner of the file or directory
Link	`l`	The container may create a hard link to the corresponding inode.

Execute Mapped	`m`	The container may map the file into memory for execution (other `mmap` operations are governed normally by read, write, and append access). This allows policy to distinguish shared libraries from executables.

BPFContain defines four capability rules which are used to specify exceptions to its default-deny policy on POSIX capabilities. The `net-bind` rule grants the ability to bind to privileged ports, the `net-raw` rule grants the ability to use raw sockets, and the `net-broadcast` rule grants the ability to broadcast and listen to multicast network traffic. The `dac-override` rule grants the ability to override discretionary access controls on files. All other POSIX capabilities are implicitly denied and may not be overridden. Note that these capability rules may not be used to grant overpermission to a container—the underlying process must already have the actual capability to use it.

Network policy in BPFContain enforces at the socket level, across various granularities. The most basic network policy consists of a single rule, `network`, which grants coarse-grained access to the entire networking stack. A specific address family and level of access may also be specified for finer-grained policy. Network policy is closely related to IPC policy, which grants or revokes permission to perform interprocess communication between containers. Support IPC mechanisms include System V IPC, signals, and Unix sockets (which must also be declared under network policy). For inter-container IPC to be valid, *both* containers must mutually declare IPC access to each other. In other words, if container A wishes to perform IPC with container B, container A must list an `ipc: B` right and container B must also list an `ipc: A` right.

4.4 Launching a BPFContain Container

To allow processes to request that they be placed into a container, BPFContain attaches a specialized eBPF program type called a `uprobe` (userspace probe) to a userspace library call, `bpfcontain_confine`. This function is a stub, whose only purpose is to trap to the uprobe. If it fails to trap the corresponding eBPF program (for example, if BPFContain has not yet loaded its eBPF programs into the kernel), the function returns `-EAGAIN` to indicate that the caller should repeat the request. Attaching a `uprobe` to a library call in this way is a typical eBPF design pattern, which effectively allows eBPF programs to make almost arbitrary extensions to the kernel's API.

When `bpfcontain_confine` traps to its corresponding uprobe, the uprobe queries the current PID and associates it with a corresponding container ID. This container parameterizes BPFContain's policy maps, allowing it to query the given container's correct policy. Subsequent forks associate newly created processes with the parent's container ID, assuring that the entire process subtree belongs to the same container. Once a process has been associated with a specific container ID, this association persists until the process exits, preventing the `bpfcontain_confine` call from being abused to transition to another container profile.

From the user's perspective, running a BPFContain container is as simple as invoking

the `bpfcontain run -n <name>` command. This command is a thin wrapper around the `bpfcontain_confine` library call discussed above. Its only purpose is to invoke this library call, check for a successful invocation (using its return value), and then execute the command defined in the corresponding container manifest. See Figure 4.2 for an overview of this process.

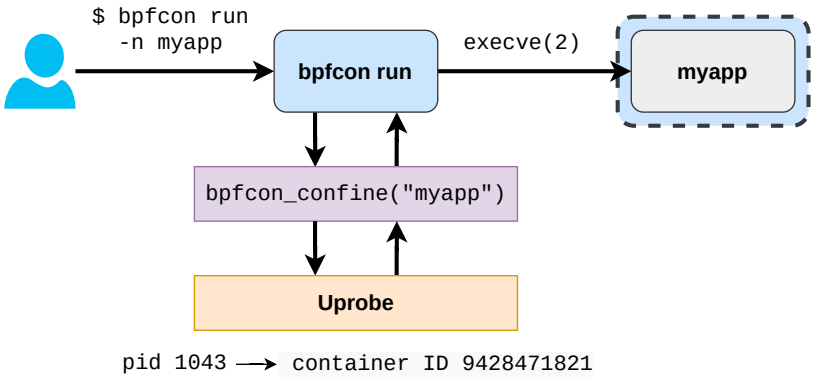

Figure 4.2: Launching a BPFContain container.

An important feature of BPFContain is that the `bpfcontain_confine` library call requires no additional operating system privileges to start confinement. This notion of unprivileged confinement is a unique advantage over other container implementations in Linux. Somewhat counter-intuitively, traditional container implementations often rely on binaries with escalated privileges (e.g. setuid root) to set up proper confinement. Failure to correctly drop these elevated privileges may result in *escalation of privilege* in the host system, particularly if the confined process manages to escape the container. By obviating this need for elevated privileges, BPFContain conforms with the principle of least privilege and improves its overall security.

As a side effect of BPFContain's design, a generic application may also simply invoke the `bpfcontain_confine` library call directly, eliminating the need to start the target application using the `bpfcontain run` wrapper. This notion of self-confinement enables application developers and package maintainers to ship BPFContain policy with their software and enforce it transparently to the end-user. Since BPFContain policy is designed to be readable and modifiable by end-users, a security policy shipped with an application could optionally be tuned by the user according to their specific needs.

4.5 Policy Enforcement Mechanism

Security policy in BPFContain falls into two categories: *implicit* and *explicit*. Explicit policy is per-container policy defined by the end-user in the container's manifest (see Section 4.3). On the other hand, implicit policy comprises sensible defaults that are applied and enforced for every container, regardless of its configuration. The vast majority of implicit BPFContain policy consists of behavioural restrictions, which prohibit a container from performing a set of dangerous actions that a container could use to escape confinement, attack other containers, or negatively impact the

host system. Conversely, other implicit policy grants a minimal set of rights to a container, including the ability to interact with its own procfs entries, communicate with other processes running in the same container, and access newly created files by processes within the container (e.g. temporary files). Table 4.3 describes each implicit policy category in BPFContain.

Table 4.3: Implicit policy in BPFContain, which is enforced regardless of a container's manifest. Implicit restrictions generally correspond with resources which a well-behaved container should never need. Implicit rights permit certain sane behaviours, such as interprocess communication between processes *within* a container. These rights effectively constitute exceptions to ordinary enforcement. Note that implicit rights may still be overridden by an explicit restriction specified in the container's manifest. A third category, implicit death, refers to accesses that cause BPFContain to send an uncatchable SIGKILL to the process.

Policy	Kind	Description
BPF	Restriction	A container is disallowed from making *any* `bpf(2)` system calls. This restriction prevents a container from loading, unloading, and accessing any eBPF programs and maps, including those which belong to BPFContain itself.
Ptrace	Restriction	A container may not use `ptrace(2)` to trace or control processes.
Kernel Lockdown	Restriction	A container is subject to full Kernel Lockdown [35] restrictions, disabling all operations that could be used for arbitrary code execution in the kernel.
Kernel Modules	Restriction	A container may not load any modules into the kernel.
Kexec	Restriction	A container may not use `kexec`-family system calls to load new kernels.
Shutdown	Restriction	A container may not shut down or reboot the system.
Key Management	Restriction	A container may not interface with the kernel's key management mechanisms.
Quotactl	Restriction	A container cannot use the `quotactl(2)` system call to bypass restrictions on resource consumption.
Rlimit	Restriction	A container cannot use the `getrlimit(2)`, `setrlimit(2)`, or `prlimit(2)` system calls to get or set resource limits.
Scheduler	Restriction	A container cannot inspect or modify process scheduling or I/O scheduling priority.
Mount	Restriction	A container cannot mount, remount, unmount filesystems or move filesystem mounts.
Pivot Root	Restriction	A container cannot pivot the root directory of a filesystem.
Syslog	Restriction	A container cannot use the `syslog(2)` system call to access the kernel logs.

Set Time	Restriction	A container cannot use the `settime(2)` system call to change the system time.
Privilege Escalation	Death	To prevent kernel privilege escalation exploits, which typically rely on forcing the execution of `commit_creds` [32], BPFContain will outright kill a contained process that invokes this function (outside of the execve code path) to try to escalate its privileges.
IPC	Right	A process can always perform interprocess communication with another process within the same container.
Procfs	Right	A process is granted full access to its own procfs entries and those belonging to other processes within the same container.
New Files	Right	A container is granted full access to any new files or directories that it creates.

Following the principle of least privilege [50], BPFContain implements strict default-deny enforcement. The user may optionally change this behaviour and elect to enforce a default-allow policy instead by setting `default: allow` in the manifest. A default-allow policy enables the easy restriction of specific unwanted behaviour in a given program without worrying about the details of constructing a rigorous security policy. Unless a container has been marked as default-allow, all access requests that are not covered under the implicit or explicit policies for a container are denied by default. Any denied access request is logged to userspace by the BPFContain daemon.

4.5.1 LSM Probes and Maps

At runtime, BPFContain stores representations of explicit policy rules using several eBPF maps. In particular, it uses two maps for each major policy category, one to store allowed access patterns and another to store denied access patterns. BPFContain uses a composite key comprised of a unique container ID combined with another unique identifier for the given resource as a key for each policy map. For instance, a filesystem policy key consists of the container ID and a unique filesystem device ID provided by the kernel. File policy keys are similar, but with the addition of an inode number to identify a specific file. Each key in a policy map maps to a vector describing the allowed or denied access pattern. BPFContain also maintains a few other eBPF maps in addition to its policy maps, used to track the state of running processes, the containers they belong to, and other aspects of the running system, such as which procfs inodes belong to which containers.

When BPFContain makes a policy decision, it considers the current container's policy along with its current state (including the state of whatever processes are running in the container). For explicit policy, BPFContain compares the requested access vector with the values stored in its policy maps. For access to be allowed, the entire access vector must be a subset of the value stored in the "allow" policy map. On the other hand, a denial occurs if any one of the accesses in the

"deny" map matches any part of the access vector. If no matching entry is found in either policy map and the container has not been marked default-allow, access is denied by default.

4.5.2 Preventing Kernel Privilege Escalation

Xin *et al.* [32] identified a common class of container privilege escalation attack which works by exploiting kernel code execution vulnerabilities to force an invocation of the kernel's `commit_creds` function. The attacker then uses this function to update their process' credentials with escalated privileges. Their original paper proposed a simple defence involving a 10 line patch to the kernel's `commit_creds` function that adds a check to see if a namespace confines the process. If it is, assume that it is in a container, and block any updates to credentials that would result in escalation of privilege [32]. While effective, this solution is inflexible in that it assumes a one-to-one correlation between namespaces and container membership, and its implementation requires an out-of-tree kernel patch.

BPFContain offers an elegant solution to this kernel privilege escalation problem by using a kprobe (kernel function probe) eBPF program, instrumenting the `commit_creds` function. Kprobes work by replacing a kernelspace address with a trap to an eBPF program; when the eBPF program returns, the kernel function proceeds with normal execution [22]. Using eBPF maps to keep track of the running process's state, BPFContain can determine whether or not the call to `commit_creds` has been gated by one of its LSM probes. If not, BPFContain will immediately terminate the offending process by sending an uncatchable SIGKILL signal from the kernel. Listing 4.1 depicts the code for the kprobe.

Listing 4.1: BPFContain's `commit_creds` kprobe. This code consists of a query to the eBPF map holding BPFContain's process states, killing the process if it is not being transformed by an execve operation (as flagged by BPFContain's LSM probes).

```
kprobe__commit_creds(struct pt_regs *ctx) {
  // Get the current PID
  u32 pid = bpf_probe_get_current_pid_tgid();

  // Look up process state (only exists if the process is in a container)
  struct bpfcon_process *process = processes.lookup(&pid);
  if (!process || !process->container_id)
    return 0;

  // Kill the offending process if it is not being transformed by an execve
  if (!process->in_execve)
    bpf_send_signal(SIGKILL);

  return 0;
}
```

This technique enables BPFContain to enforce simple control flow integrity in the kernel, preventing the privilege escalation exploit. Thanks to eBPF, BPFContain can do this at runtime without requiring a kernel patch or even a system reboot. Further, the seamless integration provided

by eBPF maps enables BPFContain to apply this enforcement exclusively on its own containers and only on invalid code paths. These factors result in a flexible yet effective solution to the kernel privilege escalation problem.

4.6 Case Study: Discord

As a motivating example of how BPFContain policy might be used in practice, consider the Discord voice chat client. Discord [13] is a popular, cross-platform voice chat client designed for gamers. It comes with an optional feature called "Display Active Game", which automatically displays whatever game the user is currently playing in their status message. Naturally, there are privacy concerns here because some users might not want to broadcast their current activity to the rest of the world; therefore, Discord provides an option to turn this feature off. However, an strace [67] of the Linux Discord client reveals that Discord continually scans the entire process tree every 5 seconds and queries the full path of every executable running on the system, even when "Display Active Game" has been turned off. This scanning behaviour represents a gross violation of an end-user's privacy expectations[3], as Discord could be, in principle, doing whatever they like with this data. For simple use cases such as this one, BPFContain provides users the ability to write a simple yet effective policy to eliminate this problematic behaviour. Listing 4.2 and Listing 4.3 depict two examples of what such a policy might look like.

Listing 4.2: A sample manifest for Discord [13] using BPFContain's more restrictive default-deny confinement. All accesses which are not listed under the container's rights are implictly denied. The explicit restriction on access to `procfs` prevents Discord from scanning the process tree, regardless of its rights.

```
name: discord
cmd: /bin/discord
rights:
  - filesystem /
  - network
  - video
  - sound
  - graphics
  # Discord needs access to these files in order to start without crashing
  - file /proc/modules r
  - file /proc/sys/kernel/yama/ptrace_scope r
restrictions:
  - filesystem /proc
```

Listing 4.3: A sample manifest for Discord [13] using BPFContain's optional default-allow confinement. This permits a much simpler policy that directly targets Discord's `procfs` scanning behaviour.

```
name: discord-allow
```

[3] See https://www.reddit.com/r/privacy/comments/coo3h6/discord_uses_ptrace_wildly and https://www.reddit.com/r/discordapp/comments/g2nwa0/how_can_i_disable_discord_from_scanning_the for examples of user frustration.

```
cmd: /bin/discord
default: allow
rights:
  # Discord needs access to these files in order to start without crashing
  - file /proc/modules r
  - file /proc/sys/kernel/yama/ptrace_scope r
restrictions:
  - filesystem /proc
```

In the first example (Listing 4.2), the container grants access to the root filesystem, networking capabilities, and video and sound devices. It explicitly restricts access to the `procfs` filesystem, preventing Discord from scanning the process tree. In the second example (Listing 4.3), a more permissive policy is defined, which only restricts access to `procfs`. The choice of which alternative to use is left entirely up to the user. It may depend on various factors such as the existence of a pre-configured policy file, the desired use case, and the user's level of comfort with BPFContain's policy semantics.

4.7 Why an eBPF Implementation?

The classical method for extending the kernel in Linux has traditionally been through kernel modules or kernel patches. While ordinary kernel modules are often loadable at runtime, they carry an inherent risk of corrupting data, crashing the kernel, or otherwise damaging a running system. Kernel patches carry essentially the same risks, except that they also require rebooting the system once installed. Like a kernel patch, traditional LSM implementations also need to be loaded into the kernel at boot time and therefore may not be updated at runtime. With eBPF and KRSI [10, 54], we can dynamically attach LSM programs at runtime, allowing dynamic modification of kernel security policy at a fundamental level. BPFContain's implementation using eBPF LSM programs means that it can be dynamically loaded and unloaded on a vanilla Linux kernel with no downtime.

Compared with kernel patches and kernel modules, eBPF provides guaranteed production safety due to its restricted execution environment and in-kernel verifier. In particular, an eBPF program is guaranteed not to crash the kernel, cause a deadlock, or access dangerous memory locations. Even in the case of vulnerabilities related to loading and running eBPF programs, these can be fixed by patching the JIT compiler and verifier without needing to modify the underlying BPF program [22]. For example, the BPF JIT compiler has been hardened against Spectre/Meltdown-style speculative execution attacks [25] through a patch that allows it to implicitly steer memory access in conditional branches into safe regions [65]. The guaranteed safety provided by eBPF programs is a significant advantage over traditional kernel extension methods. Since an eBPF program can be dynamically loaded and unloaded without negatively impacting the rest of the running system, eBPF programs can be more readily accepted in production environments [22].

Sultan *et al.* [68] discussed the importance of moving towards container-specific LSMs to enforce per-container policy. Thanks to eBPF, BPFContain constitutes such a container-specific LSM

implementation, as it can be loaded without impacting the underlying system and can dynamically apply policy on a per-container basis. As a container-specific LSM, BPFContain is virtually transparent to the rest of the system, in stark contrast with traditional LSM-based approaches such as SELinux [57] and AppArmor [11], which enforce system-wide policy.

Besides LSM programs, BPFContain also takes advantage of *other* BPF program types for additional hardening of the container-host boundary. For instance, BPFContain uses a kprobe (kernel function probe) program to dynamically probe the `commit_creds` function in the kernel responsible for updating user credentials. In combination with its LSM probes, this allows BPFContain to enforce on calls to this function *outside* of the `execve(2)` code path. Thanks to this kprobe, BPFContain can effectively stop kernel privilege escalation attacks such as those described by Xin *et al.* [32], which rely on kernel exploitation techniques to invoke the `commit_creds` function. Further, this can be done at *runtime*, without patching or rebooting the kernel. Future versions of BPFContain can use similar probes on other kernel and userspace functions to achieve even finer-grained hardening.

5 Evaluation

5.1 Security

According the Anderson's reference monitor model [1], a secure reference validation mechanism must satisfy the properties of *complete mediation*, *tamper resistance*, and *verifiability*. Complete mediation refers to the property of mediating all possible accesses to a security-sensitive resource with no ability to bypass the reference monitor. Tamper resistance means that it should not be possible for an external actor to interfere with or change the reference monitor's behaviour. Finally, verifiability refers to the ability to formally or informally verify the correctness of a reference monitor. Generally speaking, this means that the underlying implementation should be terse and simple enough that its efficacy can be reasoned about in practice [24]. As a least-privilege implementation, BPFContain must adhere to all three of these properties (at least within the scope of container security) to be considered secure. I discuss each in the paragraphs that follow.

Complete Mediation Recall that most of BPFContain's policy enforcement mechanism leverages the Linux Security Modules framework exposed by the kernel. We assume that the property of complete mediation holds for the LSM framework itself. Therefore, we can say that BPFContain achieves complete mediation insofar as its LSM-level policy is concerned. Other aspects of BPFContain's policy enforcement, such as its instrumentation of the `commit_creds` function in the kernel, serve only to complement its LSM-level policy rather than replace it. Such extensions provide additional kernel-level hardening against attacks mounted from containers and, thus, increase the strength of BPFContain's complete mediation guarantees. In summary, we can say that BPFContain's complete mediation at least reduces to that of the LSM framework itself, and

extends it in the best case.

Tamper Resistance Due to BPFContain's eBPF-based implementation, the presence of the `bpf(2)` system call on the host system introduces certain risks that must be carefully managed to achieve tamper resistance. The primary risk involves unwanted modification of BPFContain's policy maps. To mitigate this risk, BPFContain includes specific logic in its LSM probe on the `bpf(2)` system call itself, preventing any userspace process besides the BPFContain daemon itself from directly accessing or modifying its policy maps. Additionally, maps responsible for managing process and container state are restricted using the `BPF_F_READONLY` flag, meaning that they can only be modified from within BPFContain's LSM probes, and never via the `bpf(2)` system call [33]. Even without these safeguards, the kernel gates access to eBPF maps with the CAP_SYS_ADMIN capability. This requirement means that a process would effectively require root privileges before accessing any map on the system [33].

With the integrity of BPFContain's policy and state maps covered, all that remains is to show that its maps and eBPF programs are also protected from being removed or replaced in the kernel. For this, we rely on the inherent properties of how the kernel manages the lifecycle of eBPF maps and programs. For each *BPF object*[4], the kernel maintains a reference counter, keeping track of the number of open file descriptors referring to the object. The kernel only detaches and cleans up a given BPF object once its reference counter reaches zero [63]. Thus, so long as the BPFContain daemon remains running and does not close these open file descriptors, its programs and maps are guaranteed to be loaded in the kernel. As an extra precaution, BPFContain also pins all of its maps and programs to a special filesystem called `bpffs`. Pinning an object to `bpffs` increments its reference counter by one until an `unlink(2)` system call removes the pinned inode [63]. Thus, BPFContain's eBPF objects persist even when BPFContain is killed, for example, by a malicious privileged process. To protect its pinned objects, BPFContain instruments an LSM probe preventing any other process from calling `unlink(2)` on its pinned file descriptors.

Verifiability BPFContain has a comparatively small codebase (well under 2000 lines of kernelspace code) compared to conventional LSM implementations such as SELinux or AppArmor. Much of its functionality, such as policy management, is offloaded entirely to userspace, further reducing the amount of kernelspace code that must be verified. Finally, because of its eBPF implementation, BPFContain's kernelspace code must pass automated, formal verification checks before it can even be loaded into the kernel. Because of these checks, we can be confident that BPFContain's enforcement mechanism does not suffer from the memory safety issues and other security-critical bugs that plague conventional C programs. Therefore, we can say that BPFContain achieves the verifiability property.

[4]Think of this as an umbrella term covering both eBPF maps and programs.

5.1.1 Evaluating BPFContain's Security in Practice

Evaluation of BPFContain's security in practice will require demonstration that a correctly configured BPFContain policy can defend against container exploitation. This will involve three phases:

- Identifying relevant exploits for a set of container images;
- Constructing a security policy for these images; and
- Testing that the security policy effectively prevents the exploits without affecting the desired functionality of the container image.

Xin *et al.* [32] presented a rigorous dataset of container exploits up to the year 2018, which can bootstrap the testing dataset for BPFContain and serve as a strong basis for comparison against existing container security measures. This dataset will then be augmented with additional CVEs from 2019 and 2020.

In addition to container-specific exploits, we can also test BPFContain's efficacy for ad-hoc confinement by constructing relevant BPFContain policy for commodity applications and testing against known CVEs or a set of undesired behaviour for these applications. For instance, these tests might involve verifying the prevention of a code execution exploit in a web server or stopping Discord's unwanted procfs scanning behaviour using the policy depicted in Section 4.6.

5.2 Usability

Informally, the usability argument for BPFContain is obvious. By exposing a high-level policy language with semantics that will be familiar to end-users, BPFContain makes it easy to write secure policies that pertain to a container's specific needs. Higher-level rules may be combined with lower-level rules for finer-grained enforcement where required. For use cases that focus on limiting specific behaviours in a confined application, the default-allow policy option provides a convenient way to write ad-hoc policy without worrying about the underlying details. For instance, the default allow policy for restricting Discord's procfs scanning behaviour (c.f. Listing 4.3 in Section 4.6) is only nine lines long, and consists of three actual policy rules.

To properly evaluate BPFContain's usability, we must establish a strong basis for comparing it to other confinement mechanisms, container-focused and otherwise. To that end, a user study will be conducted, wherein end-users are requested to evaluate BPFContain based on its perceived security level, ease of policy authorship, and alignment with user expectations. Before providing an evaluation, a participant would use BPFContain for some time to confine commonly-used applications such as text editors, voice chat clients, and web browsers. Because BPFContain targets both ad-hoc confinement and package management as potential use cases, it may also be valuable to conduct a similar study asking application authors to write BPFContain policy for

their own software projects. Participants would be asked to provide the same evaluation for an existing policy mechanism, such as AppArmor, to establish a basis for comparison with other policy mechanisms.

5.3 Performance

To establish BPFContain's performance overhead on the running system, we can employ various benchmarking tests that target the operations BPFContain interposes on. The same benchmarks will be run on a standard Docker configuration without any confinement and a Docker configuration using Docker's basic AppArmor and seccomp policy to establish a baseline for comparison. Since Linux 5.1, the kernel now also supports runtime benchmarking of eBPF programs, providing a means of directly measuring the overhead associated with a given probe [64]. Using this data, we can isolate each eBPF program's precise overhead and corroborate the other tests' results. Table 5.1 describes the proposed benchmarks in detail. For reproducibility, tests will be conducted for multiple trials (separately including and excluding the kernel's built-in eBPF benchmarks) on a dedicated host system using a KVM virtual machine.

Table 5.1: Benchmarking tests that will be used to evaluate the performance overhead of BPFContain.

Test	Description
Phoronix OSBench	The Phoronix OSBench benchmarking suite [29] is a macro-benchmarking suite that measures basic operating system functionality, such as program execution, filesystem access, and memory allocations.
System Calls	This test will involve writing micro-benchmarking programs whose only purpose is to continually invoke system calls interposed on by BPFContain's LSM probes. Some particularly relevant system calls might be `read`, `write`, `mmap`, `clone`, and `execve`. The programs will measure the average time taken to perform each system call.
Kernel eBPF Benchmarks	Linux 5.1 added a tunable sysctl parameter for enabling system-wide benchmarking of eBPF programs [64]. Using this parameter, we can directly measure the overhead imposed by BPFContain's eBPF programs during regular system operation. Additionally, these built-in benchmarks can be used to corroborate findings in the formal tests outlined above.

6 Discussion

6.1 Limitations

Due to eBPF's safety constraints, conventional maps must be constrained to some fixed size, determined at map creation time (according to the arguments of the `bpf(2)` system call used to create the map). Since BPFContain uses eBPF maps to store per-container policy, this means that, in practice, the maximum map size bounds the number of possible rules for each policy category. This upper bound is not strictly an issue since BPFContain's current design relies on loading policy when the daemon first starts, and thus map sizes can be predetermined based on policy files' contents. However, it would be desirable to add dynamically loadable policy into future versions of BPFContain, which would require some means of handling this map size restriction at runtime.

BPFContain's policy maps would either need to be very large or be dynamically resizable at runtime in order for dynamically loadable policy to be feasible. The former option would increase BPFContain's memory overhead on the running system. While eBPF maps do support runtime allocation within a fixed size upper bound using the `BPF_NO_PREALLOC` flag, this runtime allocation comes at the cost of potential deadlocks when instrumenting critical sections in the kernel [62]. Its use is generally discouraged for this reason. Linux 5.10 and 5.11 introduce some new garbage collected map types [55, 56], which could alleviate this problem in practice. Integration with these new maps is discussed as topics for future work in Section 6.2.

Another limitation of the current research prototype is that it purely focuses on least-privilege and composability, ignoring the critical goal of virtualization that is so central to the concept of containers. As explained in Section 2.3 and Section 3.2, the initial BPFContain prototype' goal was to provide a least-privilege-first approach to containers; thus, virtualization support is left as a topic for future work. The path forward for virtualization support in BPFContain could proceed in two disparate (yet not mutually exclusive) directions: (1) the introduction of application-transparent virtualization eBPF helpers in the kernel; and (2) the direct integration of BPFContain with existing container management implementations. Section 6.2 covers both options in detail.

6.2 Future Work

Linux 5.10 and Linux 5.11 introduced `inode_local_storage` [55] and `task_local_storage` [56] eBPF map types, respectively. These new map types add garbage collection functionality tied down to the lifecycle of the underlying tasks and inodes to which they are bound. This garbage collection functionality allows for the implementation of security blobs, similar to those used by more conventional LSMs. Future iterations of BPFContain will incorporate these new map types, which should allow for significant implication of state and policy management, and pave the way for dynamically loadable BPFContain policy.

While BPFContain currently only implements least-privilege for containers, there is poten-

tial to add full support for virtualization through new eBPF helpers in the kernel. These helpers could, for instance, be used to transparently move process groups into new namespaces and cgroups or manage filesystem mounts within a mount namespace, transparently to the target application. BPFContain could integrate these helpers into its container lifecycle management probes to enforce namespace, cgroup, and mount policy automatically and transparently to the target application. Not only would this extension enable fully application-transparent namespace and cgroup management, but it would also obviate the need for the root privileges required to use some generic kernel virtualization techniques.

On the policy language side, the integration of virtualization with BPFContain's enforcement mechanism presents opportunities for streamlining the configuration and policy associated with BPFContain containers. For instance, filesystem and mount namespace rules could be combined into one explicit `mount` rule. Under a given mount rule, BPFContain would mount an overlay filesystem in the container's mount namespace and automatically allow access to this mounted filesystem in its LSM policy. This new integration would not only significantly streamline container configuration, but it would also obviate the need for complex filesystem and file rules. For example, one mount rule could replace a series of file rules specifying access to required shared libraries.

An alternative route for adding virtualization support to BPFContain would be integration with existing container management frameworks such as Docker [14], Kubernetes [26], and OpenShift [46]. In principle, such integration could extend to any other container management framework, so long as it is compliant with the OCI (Open Container Initiative) standards [37]. This integration would have similar benefits to the eBPF helpers approach outlined above. Policy generation and enforcement could be tied directly with container configuration, offering a streamlined and more secure policy.

To harden the kernel against the privilege escalation attacks described by Xin *et al.* [32], BPFContain leverages a kprobe eBPF program to instrument the kernel's `commit_creds` function. Future versions of BPFContain could apply this same technique to other security-critical code paths in the kernel to protect against *other* kernel-level exploits mounted from containers. Similar techniques (using uprobes instead of kprobes) would also work on security-critical userspace applications that might be invoked within containers. Future work will involve identifying critical code paths in the kernel and the userspace TCB (trusted computing base) and building BPFContain kprobes and uprobes to harden them against exploitation.

7 Related Work

Several researchers [32, 42, 68] have examined various aspects of the container security spanning various container management platforms and confinement mechanisms. Sultan *et al.* [68] examined the container security landscape in detail, identifying strengths, weaknesses, patterns in academic literature, and promising future work opportunities. Their recommendations included work towards

a container-specific LSM [68], which inspired the research direction for BPFContain. Xin *et al.* [32] presented a detailed taxonomy of container security exploits and analyzed container management platforms' security against their exploit database. Mullinix *et al.* [42] presented a comprehensive analysis of Docker security and its underlying mechanisms along with the state-of-the-art solutions in academia for measuring and hardening Docker security.

Vulnerability analysis of container images [6, 27, 53] has proved a lucrative technique for identifying areas of weakness in container configurations. Shu *et al.* [53] presented their DIVA framework for automatic Docker Hub image vulnerability analysis and aggregated vulnerability data from over 350,000 Docker Hub images. In particular, they found that Docker images contained an average of 180 security vulnerabilities and that these vulnerabilities often propagate between parent and child images [53]. Kwon and Lee [27] used a similar technique in DIVDS, extracting vulnerabilities from container images and offering an interface to compare vulnerability severity and optionally add specific vulnerabilities to an allowlist. Brady *et al.* [6] applied image vulnerability scanning to a continuous integration pipeline to identify container vulnerabilities in production deployments.

Other approaches consider ways to harden container management platforms directly, using existing Linux security features. Chen *et al.* [7] proposed a framework for mitigating denial of service attacks against the host system mounted from containers, using a combination of cgroups and kernel-module-based enforcement to limit resource consumption. While such cgroup-based methods effectively restrict resource-based denial of service attacks, they are insufficient for implementing least-privilege. To simplify system call filtering rules and reduce overprivileged access to the host system, Lei *et al.* [30] introduced SPEAKER to partition a container's seccomp-bpf profile into multiple execution phases. The critical insight that informed their approach was that a container's setup phase and main work loop often involve disparate sets of system calls [30]. Xin *et al.* [32] proposed patching critical functions in the kernel, such as `commit_creds` to be container-aware to mitigate the threat of kernel privilege escalation exploits mounted from containers.

Many least-privilege enforcement mechanisms for container security rely on Linux Security Modules for mandatory access control. Loukidis *et al.* [39] proposed a mechanism for automatically deriving per-container AppArmor policy based on image characteristics and runtime information gathered from individual containers. Based on the observation that different containers often have disparate security requirements, Sun *et al.* [69] proposed adopting a new security namespace to allow specific containers to load their own LSM implementations, independent of the rest of the system. Citing their work as a good starting point, Sultan *et al.* [68] proposed that further research should be dedicated to the notion of a container-specific LSM. BPFContain represents another step towards such a container-specific LSM implementation.

BPFContain is not the first research project to propose exposing LSM hooks to userspace through eBPF. Landlock [48, 49] is an experimental Linux Security Module presented by Salaun to expose a subset of LSM hooks to unprivileged userspace programs. Under Landlock, userspace programs write and load eBPF programs into the kernel to filter their accesses. Unfortunately, the

community has since recognized that allowing unprivileged processes to load eBPF programs into the kernel is fundamentally insecure, regardless of any limitations imposed on program type and functionality [10]. Thus, Landlock has not been merged into the mainline kernel and will likely remain out-of-tree going forward. Unlike Landlock, Singh's KRSI [10, 54] allows privileged users to attach eBPF programs to LSM hooks. Since KRSI does not require unprivileged processes to load and manage eBPF programs, it does not suffer the same fundamental security issues that have detracted from Landlock.

While KRSI serves as the infrastructure for implementing LSM programs in eBPF, developers must still provide their own implementations for any eBPF LSM hooks they wish to use. Findlay *et al.* [17] introduced bpfbox as the first full process confinement mechanism using these eBPF LSM hooks. Unlike BPFContain, bpfbox was focused on a generic sandboxing use case rather than a container-specific LSM implementation. BPFContain can be thought of as a direct successor to bpfbox, taking lessons learned from its development as a generic sandboxing framework and applying them specifically to the realm of containers.

8 Conclusion

This paper has presented BPFContain, a novel least-privilege implementation for container security that leverages the power of eBPF to provide safe and flexible policy enforcement at the container level. Due to its implementation as an eBPF-based enforcement mechanism, BPFContain both hardens the kernel against privilege escalation exploits and secures the container-host boundary against undesired (and potentially dangerous) misuse of system resources. Further, BPFContain exposes a simple YAML-based policy configuration language to userspace that conforms to existing container management mechanisms' semantics. This policy language is designed to be highly configurable and to support ad-hoc confinement use cases through high-level policy rules and optional default-allow enforcement.

In the future, formal evaluation is necessary to establish BPFContain's usability, security, and performance overhead. Direct comparison against other confinement mechanisms is essential to determine BPFContain's relative efficacy and encourage its adoption as a new standard for secure containers. Other promising avenues for future work include the addition of full virtualization support, direct integration with OCI-compliant container management systems, and further investigation into how else we can harden the kernel using kprobe-based instrumentation of critical code paths.

9 Acknowledgements

The idea for BPFContain was conceived during a discussion with Anil Somayaji and further developed during a discussion with Anil Somayaji and David Barrera. The basis for BPFContain

was developed during our earlier research on bpfbox [17].

References

[1] J. P. Anderson, "Computer Security Technology Planning Study," Air Force Electronic Systems Division, Hanscom AFB, Bedford, MA, Tech. Rep. ESD-TR-73-51, 1972. [Online]. Available: http://seclab.cs.ucdavis.edu/projects/history/papers/ande72.pdf.

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

[3] Lynn Erla Beegle, "Rootkits and Their Effects on Information Security," *Information Systems Security*, vol. 16, no. 3, pp. 164–176, 2007. DOI: 10.1080/10658980701402049.

[4] Oren Ben-Kiki, Clark Evans, and Ingy döt Net, *YAML Ain't Markup Language (YAML™) Version 1.2*, YAML specification. [Online]. Available: https://yaml.org/spec/1.2/spec.html (visited on 11/29/2020).

[5] Andrew Berman, Virgil Bourassa, and Erik Selberg, "TRON: Process-Specific File Protection for the UNIX Operating System," in *Proceedings of the USENIX 1995 Technical Conference*, The USENIX Association, 1995, pp. 165–175. [Online]. Available: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.56.9149&rep=rep1&type=pdf.

[6] Kelly Brady, Seung Moon, Tuan Nguyen, and Joel Coffman, "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.

[7] Jiyang Chen, Zhiwei Feng, Jen-Yang Wen, Bo Liu, and Lui Sha, "A Container-Based DoS Attack-Resilient Control Framework for Real-Time UAV Systems," in *Design, Automation & Test in Europe Conference & Exhibition*, IEEE, 2019, pp. 1222–1227. DOI: 10.23919/DATE.2019.8714888.

[8] Jonathan Corbet, "A Bid to Resurrect Linux Capabilities," *LWN.net*, 2006. [Online]. Available: https://lwn.net/Articles/199004/.

[9] Jonathan Corbet, "File-Based Capabilities," *LWN.net*, 2006. [Online]. Available: https://lwn.net/Articles/211883/.

[10] Jonathan Corbet, "KRSI — the other BPF security module," *LWN.net*, Dec. 2019. [Online]. Available: https://lwn.net/Articles/808048/.

[11] Crispin Cowan, Steve Beattie, Greg Kroah-Hartman, Calton Pu, Perry Wagle, and Virgil Gligor, "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.

[12] Alexander Crowell, Beng Heng Ng, Earlence Fernandes, and Atul Prakash, "The Confinement Problem: 40 Years Later," *Journal of Information Processing Systems*, vol. 9, no. 2, pp. 189–204, 2013. DOI: 10.3745/JIPS.2013.9.2.189.

[13] Discord, *Discord Privacy Policy*. [Online]. Available: https://discord.com/privacy (visited on 10/25/2020).

[14] Docker, *Docker Security*, 2020. [Online]. Available: https://docs.docker.com/engine/security/security (visited on 10/25/2020).

[15] Will Drewry, "Dynamic seccomp policies (using BPF filters)," Kernel patch, 2012. [Online]. Available: https://lwn.net/Articles/475019/.

[16] Jake Edge, "Another Union Filesystem Approach," *LWN.net*, 2010. [Online]. Available: https://lwn.net/Articles/403012/ (visited on 12/13/2020).

[17] 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.

[18] Flatpak, *Sandbox Permissions*, 2020. [Online]. Available: https://docs.flatpak.org/en/latest/sandbox-permissions.html (visited on 10/25/2020).

[19] FreeBSD, *bpf(4)*, BSD Kernel Interfaces Manual. [Online]. Available: https://www.unix.com/man-page/FreeBSD/4/bpf (visited on 12/13/2020).

[20] Ian Goldberg, David Wagner, Randi Thomas, and Eric Brewer, "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.

[21] Google, *Android Security Features*, Android security documentation. [Online]. Available: https://source.android.com/security/features (visited on 10/26/2020).

[22] Brendan Gregg, *BPF Performance Tools*. Addison-Wesley Professional, 2019, ISBN: 0-13-655482-2.

[23] IOVisor, *iovisor/bcc*, GitHub repository. [Online]. Available: https://github.com/iovisor/bcc (visited on 12/13/2020).

[24] Trent Jaeger, "Reference Monitor," in *Encyclopedia of Cryptography and Security, 2nd Ed*, Springer, 2011, pp. 1038–1040. DOI: 10.1007/978-1-4419-5906-5_646.

[25] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," in *IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 1–19. DOI: 10.1109/SP.2019.00002.

[26] Kubernetes, *Kubernetes*, 2020. [Online]. Available: https://kubernetes.io (visited on 11/30/2020).

[27] 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.

[28] 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.

[29] Michael Larabel and Matthew Tippett, *Phoronix Test Suite*, 2011. [Online]. Available: http://www.phoronix-test-suite.com (visited on 12/23/2020).

[30] Lingguang Lei, Jianhua Sun, Kun Sun, Chris Shenefiel, Rui Ma, Yuewu Wang, and Qi Li, "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.

[31] libbpf contributors, *libbpf*, GitHub repository. [Online]. Available: https://github.com/libbpf/libbpf (visited on 12/13/2020).

[32] Xin Lin, Lingguang Lei, Yuewu Wang, Jiwu Jing, Kun Sun, and Quan Zhou, "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.

[33] Linux, *bpf(2)*, Linux Programmer's Manual. [Online]. Available: https://www.man7.org/linux/man-pages/man2/bpf.2.html (visited on 12/13/2020).

[34] Linux, *capabilities(7)*, Linux User's Manual. [Online]. Available: https://linux.die.net/man/7/capabilities.

[35] Linux, *kernel_lockdown(7)*, Linux programmer's manual. [Online]. Available: https://man7.org/linux/man-pages/man7/kernel_lockdown.7.html (visited on 12/13/2020).

[36] Linux, *Seccomp BPF (SECure COMPuting with filters)*, Linux kernel documentation. [Online]. Available: https://static.lwn.net/kerneldoc/userspace-api/seccomp_filter.html (visited on 10/27/2020).

[37] Linux Foundation, *Open Container Initiative*, 2020. [Online]. Available: https://opencontainers.org (visited on 12/20/2020).

[38] LLVM, *BPF Directory Reference*, Developer documenation. [Online]. Available: https://llvm.org/doxygen/dir_b9f4b12c13768d2acd91c9fc79be9cbf.html (visited on 12/13/2020).

[39] Fotis Loukidis-Andreou, Ioannis Giannakopoulos, Katerina Doka, and Nectarios Koziris, "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.

[40] Karl MacMillan, "Madison: A new approach to policy generation," in *SELinux Symposium*, 2007. [Online]. Available: http://selinuxsymposium.org/2007/papers/08-polgen.pdf.

[41] 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.

[42] Samuel P. Mullinix, Erikton Konomi, Renee Davis Townsend, and Reza M. Parizi, "On Security Measures for Containerized Applications Imaged with Docker," *CoRR*, vol. abs/2008.04814, 2020. [Online]. Available: https://arxiv.org/abs/2008.04814.

[43] OpenBSD, *bpf(4)*, Device Drivers Manual. [Online]. Available: https://man.openbsd.org/bpf (visited on 12/13/2020).

[44] OpenBSD, *pledge(2)*, System Calls Manual. [Online]. Available: https://man.openbsd.org/pledge.2.

[45] Pradeep Padala, "Playing with ptrace, Part I," *Linux Journal*, vol. 2002, no. 103, p. 5, 2002. [Online]. Available: https://www.linuxjournal.com/article/6100.

[46] Red Hat, *OpenShift*, 2020. [Online]. Available: https://www.openshift.com (visited on 12/20/2020).

[47] RedSift, *redsift/redbpf*, GitHub repository. [Online]. Available: https://github.com/redsift/redbpf (visited on 12/13/2020).

[48] 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).

[49] Mickael Salaun, *landlock.io*, 2020. [Online]. Available: https://landlock.io (visited on 12/17/2020).

[50] J. H. Saltzer and M. D. Schroeder, "The Protection of Information in Computer Systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975. DOI: 10.1109/PROC.1975.9939.

[51] 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. [Online]. Available: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.963.860&rep=rep1&type=pdf.

[52] Seccomp, *libseccomp*. [Online]. Available: https://github.com/seccomp/libseccomp (visited on 10/27/2020).

[53] 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.

[54] 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/.

[55] KP Singh, "bpf: Implement bpf_local_storage for inodes," Kernel patch, 2020. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/bpf/bpf-next.git/commit/?id=8ea636848aca35b9f97c5b5dee30225cf2dd0fe6.

[56] KP Singh, "bpf: Implement task local storage," Kernel patch, 2020. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/bpf/bpf-next.git/commit/?id=4cf1bc1f10452065a29d576fc5693fc4fab5b919.

[57] 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.

[58] Justin R. Smith, Yuichi Nakamura, and Dan Walsh, *audit2allow(1)*, Linux user's manual. [Online]. Available: http://linux.die.net/man/1/audit2allow.

[59] Snapcraft, *Security Policy and Sandboxing*, 2020. [Online]. Available: https://snapcraft.io/docs/security-sandboxing (visited on 10/25/2020).

[60] 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.

[61] Ray Spencer, Stephen Smalley, Peter Loscocco, Mike Hibler, Dave G. Andersen, and Jay Lepreau, "The Flask Security Architecture: System Support for Diverse Security Policies," in *Proceedings of the 8th USENIX Security Symposium*, USENIX Association, 1999. [Online]. Available: https://www.usenix.org/conference/8th-usenix-security-symposium/flask-security-architecture-system-support-diverse-security.

[62] Alexei Starovoitov, "bpf: hash map pre-alloc," Kernel patch, 2016. [Online]. Available: https://lwn.net/Articles/679074/.

[63] Alexei Starovoitov, *Lifetime of bpf objects*, Facebook, 2018. [Online]. Available: https://facebookmicrosites.github.io/bpf/blog/2018/08/31/object-lifetime.html (visited on 12/22/2020).

[64] Alexei Starovoitov, "bpf: enable program stats," Kernel patch, 2020. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/bpf/bpf-next.git/commit/?id=492ecee892c2a4ba6a14903d5d586ff750b7e805.

[65] Alexei Starovoitov, "Safe Programs. The Foundation of eBPF," in *eBPF Summit 2020*, Keynote talk, Cilium, Virtual Event, 2020. [Online]. Available: https://ebpf.io/summit-2020.

[66] 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.

[67] Strace, *strace: linux syscall tracer*, Official strace website. [Online]. Available: https://strace.io (visited on 11/29/2020).

[68] 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.

[69] Yuqiong Sun, David Safford, Mimi Zohar, Dimitrios Pendarakis, Zhongshu Gu, and Trent Jaeger, "Security Namespace: Making Linux Security Frameworks Available to Containers," in *27th USENIX Security Symposium*, USENIX Association, 2018, pp. 1423–1439. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/sun.

[70] US Department of Defense, "Trusted Computer System Evaluation Criteria," DOD Standard DOD 5200.58-STD, 1983.

[71] 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.

[72] Robert N. M. Watson and Jonathan Anderson, *capsicum(4)*, FreeBSD user's manual. [Online]. Available: https://www.unix.com/man-page/freebsd/4/capsicum/.

[73] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway, "Capsicum: Practical Capabilities for UNIX," in *Proceedings of the 19th USENIX Security Symposium*, USENIX Association, 2010, pp. 29–46. [Online]. Available: https://www.usenix.org/legacy/event/sec10/tech/full_papers/Watson.pdf.

[74] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman, "Linux Security Modules: General Security Support for the Linux Kernel," in *Proceedings of the 11th USENIX Security Symposium*, USENIX, 2002, pp. 17–31. [Online]. Available: http://www.usenix.org/publications/library/proceedings/sec02/wright.html.