BPFContain: Fixing the Soft Underbelly of Container Security

William Findlay
Carleton University

David Barrera
Carleton University

Anil Somayaji
Carleton University

arXiv:2102.06972v1 [cs.CR] 13 Feb 2021

Abstract

Linux containers currently provide limited isolation guarantees. While containers separate namespaces and partition resources, the patchwork of mechanisms used to ensure separation cannot guarantee consistent security semantics. Even worse, attempts to ensure complete coverage results in a mishmash of policies that are difficult to understand or audit. Here we present BPFContain, a new container confinement mechanism designed to integrate with existing container management systems. BPFContain combines a simple yet flexible policy language with an eBPF-based implementation that allows for deployment on virtually any Linux system running a recent kernel. In this paper, we present BPFContain's policy language, describe its current implementation as integrated into docker, and present benchmarks comparing it with current container confinement technologies.

1 Introduction

Linux containers have become the preferred unit of application management in the cloud, forming the foundation of Docker [11], Kubernetes [20], Snap [46], Flatpak [14], and others. By including just the binaries, libraries, and configuration files needed by an application, containers enable simplified deployment of vendor-packaged applications, rapid horizontal application scaling, and direct developer-to-production DevOps workflows, all without the overhead of hypervisor-based virtual machines (HVMs).

Many have recognized that containers currently offer weak isolation guarantees [25, 49, 50]. Weak container confinement is less of a risk when all containers on a given host are deployed by the same party. However, strong container isolation mitigates privilege escalation attacks and is a critical requirement in multi-tenancy environments where attacker-controlled containers co-exist with benign containers.

The key to improving container isolation lies in recognizing that isolation is not the same as virtualization. Virtualization can exhibit isolation characteristics as a side effect; however, such isolation is rarely enough on its own to serve a security barrier. In networking, network address translation (NAT) virtualizes IP addresses so one IP address can be shared by an entire network of systems. While this many-to-one mapping provides a significant degree of isolation to systems behind the NAT, it is no substitute for a network firewall, especially one that is configured, e.g., to block outbound connections. Linux containers are virtualized using cgroups [5] and namespaces [18], while confinement is enforced through seccomp-bpf [29] and mandatory access control mechanisms such as SELinux [45] and AppArmor [10]. While these confinement mechanisms are powerful and flexible, container isolation was not their primary design goal, and currently they can only accomplish the task with complex policies that are difficult to write and audit.

To properly isolate containers, the kernel requires clear rules about what interactions are permissible, whether between containers or with the host OS. Much like firewall rules specify which packets may traverse it, OSes need unambiguous, auditable rules to determine what kernel-level operations are and are not allowed based on container boundaries. Following Unix tradition, the Linux kernel provides only security mechanisms and abstractions for defining security policies, but leaves policies themselves to be managed in userspace. However, unlike processes, files, and network connections, the Linux kernel has no unified abstraction around containers.

We assert that the lack of strong isolation guarantees for containers arises from a semantic gap between the security mechanisms that currently exist in the Linux kernel and the policies that we wish to define to isolate containers. As long as the kernel does not implement container-level access control mechanisms, the result will be complex, circumventable container isolation.

The Linux kernel has recently gained an alternative way to implement security abstractions: eBPF. An extension of the Berkeley Packet Filter [33], Linux's eBPF now allows for complex monitoring and manipulation of kernel-level events. Thanks to implicit load-time verification of eBPF bytecode,

eBPF provides strong performance, portability, and safety guarantees. More recent improvements to eBPF [9] have enabled interfacing with Linux Security Module (LSM) hooks, allowing eBPF code to implement new kernel-level security mechanisms.

This paper proposes BPFCONTAIN, a novel approach to container security under the Linux kernel, rectifying overprivileged and insecure containers. Leveraging eBPF, BPFCONTAIN uses runtime security instrumentation to implement container-aware policy enforcement and harden the host kernel against privilege escalation attacks mounted from within containers. Specifically, BPFCONTAIN attaches eBPF programs to LSM hooks and critical functions within the kernel to enforce per-container policy in kernelspace.

Thanks to eBPF's dynamic instrumentation capabilities, this integration occurs at runtime and requires no modification or patching of the kernel. BPFCONTAIN addresses the container userspace/kernelspace semantic gap by defining a new YAML-based policy language for container confinement that allows for simple default deny and default allow policies, high-level semantically meaningful permissions, and (where necessary) fine-grained control over the LSM API, all at the level of containers.

While BPFCONTAIN can co-exist with seccomp-bpf, SELinux, and other existing Linux security mechanisms, it does not rely on them, and in fact in the context of container isolation, BPFCONTAIN makes them redundant. It also has modest userspace requirements, consisting only of a small daemon and a control program, both of which are written in Rust. In summary, we make the following contributions:

- We present the design, implementation, and evaluation of BPFCONTAIN, a container-aware security enforcement mechanism for Linux. BPFCONTAIN is available[1] under a GPLv2 license, and installation requires only a 5.10 or newer Linux kernel. While there is past work on using eBPF for process sandboxing [13], BPFCONTAIN is both more general and more deployable, implementing mechanisms and a policy language specific to container confinement and built using tools such as BPF CO-RE and Rust that have much lower space and runtime overhead.

- We design a flexible policy language for confining containers that offers optional layers of granularity to meet a wide range of real world container use cases. The policy language is YAML-based, and expressive enough that it can be used to confine individual system resources, yet simple enough that it affords ad-hoc confinement use cases.

- We discuss integrating BPFCONTAIN with existing container management frameworks without modifying their source code, offering significant security advantages over traditional approaches to container isolation and least-privilege.

[1] https://github.com/willfindlay/bpfcontain-rs

The rest of this paper proceeds as follows. Section 2 presents background and our motivation for implementing an eBPF-based container security mechanism. Section 3 presents an overview of BPFCONTAIN and discusses our threat model and design goals. Section 4 describes BPFCONTAIN's policy language. Section 5 discusses the design and implementation of its userspace components and enforcement engine. Section 6 presents an evaluation of BPFCONTAIN's security and performance. Section 7 covers related work and Section 8 discusses limitations and opportunities for future work. Section 9 concludes.

2 Background and Motivation

This section reviews current techniques for achieving virtualization, isolation, and least privilege on Linux containers. It also provides background on the classic and extended Berkeley Packet Filters (BPF and eBPF), and discusses the motivation behind a new container-focused security enforcement mechanism.

A container is a userspace representation of a set of processes that share the same virtualized view of files and system resources. Containers run directly on the host operating system and share the underlying OS kernel, and thus do not require a full guest operating system or hypervisor to provide the virtualized view of the system to applications (see Figure 2.1). To support security beyond basic process isolation, modern container runtimes rely on a variety of low-level Linux kernel facilities to enforce virtualization, isolation, and least-privilege. Note that by relying on a disparate set of mechanisms and corresponding policies, a failure in the enforcement or policy in any single mechanism exposes the container to attacks.

2.1 Container Security

Namespaces and Control Groups (cgroups). These mechanisms 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. Cgroups (control groups) limit available *quantities* of system resources, such as CPU, memory, and block device I/O. Namespaces and cgroups on their own cannot enforce fine-grained access policies (e.g., to resources within a namespace), and thus require additional mechanisms as described below.

POSIX Capabilities. These provide a finer-grained alternative to the all-or-nothing superuser privileges required by

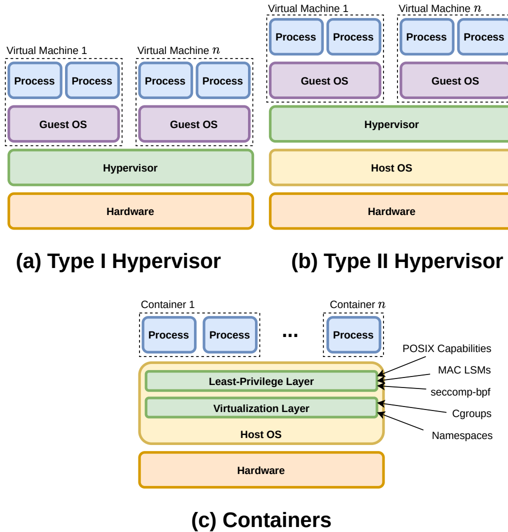

Figure 2.1: Comparison of virtual machine and container architectures. Type I hypervisors **(a)** virtualize and control the underlying hardware directly, but require full guest operating systems on top of the virtualization layer. Type II hypervisors **(b)** run on top of a host operating system but 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.

certain applications[7, 8, 27]. POSIX capabilities can be used to grant limited 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 does. POSIX capabilities add complexity to the already complex Linux permission model [7, 8]. It is challenging to create a default set of capabilities that work for most use cases. For example, Docker provides containers with 15 Linux capabilities by default, including `CAP_DAC_OVERRIDE`, which allows a container to override all discretionary access control checks [11, 49].

System Call Filtering. Linux's seccomp-bpf [29] is a common approach for reducing the set of system calls available to an application. Container runtimes will often ship with a seccomp-bpf policy that prevents containers from making calls that are known to be dangerous. Despite the high degree of control that seccomp-bpf offers to applications, it is not without its own usability and security concerns. 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)`.

Mandatory Access Control (MAC). The Linux Security Modules (LSM) API [51] provides an extensible security framework for the Linux kernel, allowing for the implementation of kernelspace security mechanisms that can be chained together. SELinux [45] and AppArmor [10] use the LSM API to provide fine-grained mandatory access control throughout the system. Container runtimes often ship default MAC policies for their containers, and requesting enforcement if such a MAC system is enabled. To our knowledge, no container runtime will refuse to run if no LSM is loaded, effectively failing open. MAC policies are also known to be difficult to maintain and audit [42].

2.2 Classic and Extended BPF

The original Berkeley Packet Filter (BPF) [33], hereafter referred to as Classic BPF, was a packet filtering mechanism implemented initially for BSD Unix. Classic BPF implemented a simple register-based virtual machine language and efficient buffer data structures to minimize the context switches while making filtering decisions. As an efficient packet filtering mechanism, Classic BPF quickly gained traction in the Unix community and has since been ported to many Unix-like operating systems, most notably Linux [26], OpenBSD [36], and FreeBSD [15]. On Linux, the seccomp sandboxing facility has been extended to use BPF to make security decisions about system calls in a confined process (c.f., seccomp-bpf [12, 29]).

A complete rewrite of the Linux BPF engine, dubbed Extended BPF (eBPF), was merged into the mainline kernel [48] in 2014. 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, 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 system introspection and kernel extension capabilities. eBPF programs execute in the kernel 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 [16]. 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 [16].

A privileged userspace process may load an eBPF program into the kernel using Linux's `bpf(2)` system call (see Figure 2.2). While it is possible to write eBPF bytecode by hand [16], several front-ends exist for compiling eBPF bytecode

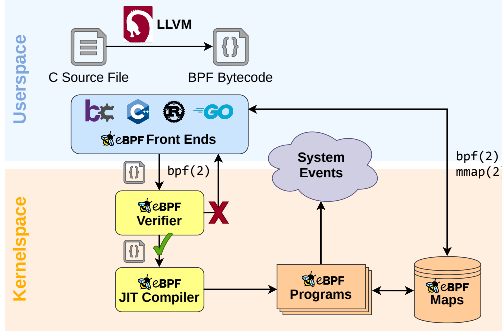

Figure 2.2: Linux kernel eBPF architecture.

from a restricted subset of the C programming language[2], including bcc [17] and libbpf [24]. These front-ends typically use the LLVM [31] 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.

2.3 Motivation

Isolation has not been a focus of container management frameworks, with many taking a lax attitude towards least-privilege enforcement [49]. Popular container frameworks such as Docker [11] provision overly-permissive default access, rely on a complex and often ill-suited suite of security mechanisms provided by the host system (see Section 2.1), and support insecure configuration options [11, 13, 25, 49]. Again, the main challenge in providing strong container isolation is that the kernel has no unified abstraction to represent containers, which results in a patchwork of security mechanisms (see Section 2.1) each enforcing specific policies. A vulnerability in any individual mechanism, or a misconfiguration in any individual policy opens up the container or system to attack.

We argue that the key to providing strong container isolation is to bridge the semantic gap between the kernel security enforcement mechanisms and container-level security policies. One approach is to extend the Linux kernel with a container-aware LSM, much like AppArmor or SELinux (see Section 2.1). However, maintaining out-of-tree kernel modules is challenging, as they must be continually updated as the kernel evolves. Another issue is adoption; users may be reluctant to run a third-party module because bugs can cause system crashes or data loss.

In the case of container isolation the situation is even worse, as a single kernel-level security mechanism is unlikely to work for all use cases. Some users (e.g., operators of multi tenant container clouds) will want strong isolation for their containers, while others (e.g., end users) will want containers to deeply integrate with the resources provided by other containers and the host system.

Containers have strong conceptual backing in userspace, but lack a unifying abstraction in the kernel. From the kernel's perspective, a containerized process is just like any other—it may be running under a different set of namespaces or in a different control group, but there exists no unifying definition of precisely what a container is, from the kernel's perspective. This lack of a solid container abstraction in turn widens the semantic gap between per-container security policy authored in userspace and policy enforcement in the kernel. We argue that BPFCONTAIN can provide such a unifying abstraction for policy enforcement. Since BPFCONTAIN requires no modification to the host kernel and can be dynamically loaded at runtime, we can provide such an abstraction without necessarily sacrificing forward compatibility with other approaches.

In summary, BPFCONTAIN came out of the realization that 1) with eBPF, we had the technology to implement specialized security abstractions and that 2) container confinement was a problem that could use a problem-specific security abstraction, given the complexity of existing solutions.

3 BPFCONTAIN

To confine applications, BPFCONTAIN leverages eBPF [48], a Linux technology that allows privileged userspace processes to make verifiably safe runtime extensions to the operating system kernel. A privileged userspace daemon manages BPFCONTAIN's eBPF components and loads policy into the kernel. When a user wishes to confine an application, they invoke a shim wrapper command, which associates itself with a specific BPFCONTAIN policy and subsequently executes the target application. Figure 3.1 illustrates this process at a high level.

BPFCONTAIN associates a confined application and all its children with a universally unique ID, called the "container ID", which is used to track various information, including namespace membership and policy association. eBPF maps store per-container information and security policies, which are queried at runtime by eBPF programs responsible for making policy decisions. In this way, BPFCONTAIN makes the kernel *container-aware* and provides a mechanism for enforcing strong least-privilege guarantees at the granularity of individual containers.

[2] In principle, this language need not be C. For instance, a framework exists for writing eBPF programs in pure Rust [38]. However, C is a popular choice since it is tightly coupled with the underlying implementation of the kernel.

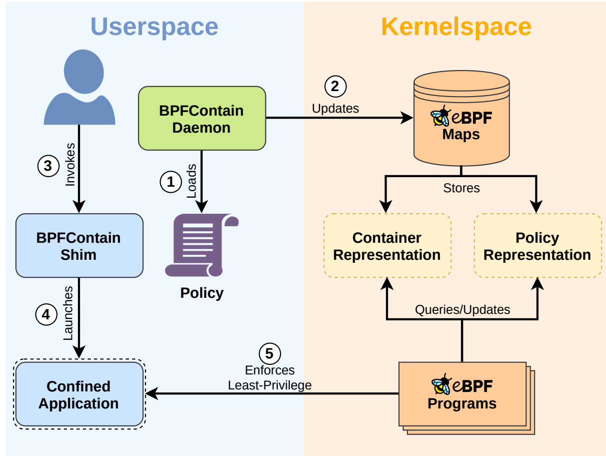

Figure 3.1: A high-level depiction of how users can use BPFCONTAIN to enforce least privilege on an application. In userspace, a privileged daemon loads parses policy files and loads policy into the kernel. The user invokes a wrapper command to launch the confined application. In the kernel, eBPF maps track the state of confined applications and active policy. eBPF LSM programs then enforce this policy when a confined application requests access to a mediated system resource.

3.1 Threat Model

BPFCONTAIN aims to protect against two categories of container-related attacks: (1) inter-container attacks, where one container attempts to interfere with or take over another; and (2) containers attacking the host, either by escaping confinement or launching denial of service or resource consumption attacks. Intra-container attacks, where an application inside a container might attack the container itself are not our focus, and are better addressed by deploying existing security mechanisms (see Section 2.1) inside the container. Host-to-container attacks are also out of scope, as they likely require the use of hardware security mechanisms [6, 25, 49].

We consider an attacker who can create and deploy containers to a multi-tenant shared container cloud. While the attacker has full control over the contents of each container they create, the attacker does not have administrative privileges on the host. The attacker's goal is to attack co-located containers or the host system. Such attacks can have a variety of goals including information extraction, application compromise, or denial of service.

We assume that an attacker-controlled process must make use of kernel interfaces (i.e., system calls) in order to mount an attack against other containers or the host system. We also assume that appropriate resource bounds are ensured through proper cgroups configuration in order to prevent resource-based denial of service attacks. Also, attacks targeting the hardware (such as Spectre [19] and Rowhammer [35]) are outside the scope of BPFCONTAIN.

While this threat model might appear limited, it is the same one assumed by standard Linux kernel security mechanisms including SELinux and AppArmor.

3.2 Design Goals

We followed five design goals when designing both BPFCONTAIN's policy language and enforcement engine. These goals are enumerated below. Note that our focus is on isolation rather than virtualization of system resources, and BPFCONTAIN uses existing kernel facilities for virtualization. We discuss potential avenues for providing standalone virtualization capabilities in BPFCONTAIN in Section 8.1.

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 [41] by default. It should be easy to tune a BPFCONTAIN policy to respond to new threats or configurations.

Simplicity. The BPFCONTAIN policy language should be simple enough for ad-hoc use cases without sacrificing security. In pursuit of this goal, our policy language takes inspiration from other high-level policy languages for containerized applications, such as Snapcraft [46]. We also base our policy language syntax on the YAML specification [3], which is both well-understood and widely-used as a configuration language.

Flexibility. When studying typical use cases for container technology, it quickly became apparent that there exists a dichotomy in the way containers are used in practice. In industry, containers tend to be used to deploy composable micro-services, particularly in cloud contexts [1]. On the other hand, they are often used to isolate desktop applications [14, 46]. In light of these findings, we designed BPFCONTAIN to work in *both* use cases, providing a means of securing micro-services and restricting desktop applications' behaviour.

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.

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.

4 BPFContain Policy

BPFContain exposes a YAML-based policy configuration language to system administrators. By default, the BPFContain daemon loads policy from `/var/lib/bpfcontain/policy`, although this setting can be changed via an environment variable when running the daemon. At a minimum, a policy file consists of a unique *name* (later used to compute a unique identifier for the policy) and a default *entrypoint*, i.e. a pathname to an executable along with optional arguments. For instance, consider a minimal policy file for a simple, statically linked program that reads from standard in and writes to standard out. This policy is depicted in Listing 4.1.

Listing 4.1: A minimal BPFContain policy file for a statically linked application that reads from stdin and writes to stdout.

```
name: hello_minimal
entry: /usr/bin/hello.static
allow:
  # Grant read and write access to
  # /dev/tty* and /dev/pts/* devices
  - tty: rw
```

In accordance with the principle of least privilege [41], BPFContain policy defaults to a default-deny. This means that, with no additional information, the policy depicted in Listing 4.1 would deny access to *all* security-sensitive resources on the system, including regular files, directories, kernel interfaces such as character devices and special filesystems, network communication, interprocess communication, and POSIX capabilities. In other words, any process tied to this policy file would only be able to perform basic computational tasks, with no access to any resources gated by the operating system's reference monitor.

However, it is possible to specify that a default-allow policy be enforced instead. In this way, end-users can write a simple policy restricting *specific* application behaviour without worrying about the details of writing a rigorous security policy, should they so choose. For instance, a user might wish to restrict an application's access to a specific subset of kernel interfaces without worrying about other access control decisions such as access to shared libraries. Since default-allow enforcement is strictly an opt-in feature, we can support such flexible confinement without exposing an unsuspecting user to any additional risk. Configuring our above example to be default allow is as simple as adding a new line with `default: allow`.

BPFContain supports three main categories of policy rules. *Allow rules* grant access to system resources, *deny rules* restrict access to system resources, and *taint rules* specify conditions under which a BPFContain container should become *tainted*. When a security policy specifies taint rules, the resulting container is considered *untainted* until the taint rule is matched. Untainted containers are exempted from default-deny enforcement, meaning that it is possible to define a security policy that behaves *as if it were default allow* during some predetermined setup phase. A policy without any taint rules is assumed to be tainted by default. Once tainted, it is impossible for a process to become untainted.

The concept of tainting both greatly simplifies and hardens the resulting security policy. This stems from the observation that the initial setup phase and main work loop of a given application often have totally disparate access patterns—for instance, a dynamically linked C application will map shared libraries into executable memory during its setup phase, but is unlikely to perform similar mappings for the remainder of its lifecycle. Eliminating these initial access patterns from security policy simplifies policy authorship while simultaneously preventing such access patterns once a process enters its main work loop. Using this notion of tainting, we can consider a revised policy for a dynamically compiled version of our simple application, depicted in Listing 4.2.

Listing 4.2: A BPFContain policy file for a dynamically linked application that reads from stdin and writes to stdout. Policy enforcement begins after the first read from stdin (potentially untrusted user input). Note that our taint rule allows us to skip over boilerplate policy provisioning access to shared libraries.

```
name: hello_taint
entry: /usr/bin/hello.dynamic
allow:
  # Grant read and write access to
  # /dev/tty* and /dev/pts/* devices
  - tty: rw
taint:
  # Taint after reading from
  # /dev/tty* or /dev/pts/* devices
  - tty: r
```

4.1 Filesystem Policy Rules

BPFContain policy restricts access to files using a variety of rule types, each with varying degrees of granularity. The most basic, the *filesystem rule*, grants access at the granularity of a given filesystem, rooted at the provided mountpoint. *Subdir rules* grants recursive access to files rooted at a given directory, with a hard limit of 8 nested subdirectories. This hard limit is a technical limitation associated with the eBPF implementation, but can be adjusted at compile time. *File rules* grant access at the granularity of individual files. *Device rules* grant access at the granularity of commonly-used (and unprivileged) character devices, such as TTYs and the kernel's random number facilities. The policy may optionally explicitly define a specific access pattern for each filesystem object, specified using a string of access flags. For instance, read and append access to a log file would be specified as `file: /var/log/mylog.log ra`.

In addition to these explicit rules, BPFContain enforces an implicit filesystem policy on all containers. For instance,

BPFCONTAIN heavily restricts access to the procfs filesystem, which exposes per-process information along with certain interfaces into the kernel. A container may only access its *own* per-process entries in procfs and is prohibited from accessing *any* other files in procfs unless such access is explicitly marked in the container's policy file. The sysfs filesystem, which provides a similar interface into various aspects of the kernel, receives similar treatment. Overlay filesystems also receive special treatment; a container is always granted full access to an overlay filesystem belonging to its own mount namespace. In practice, this can greatly simplify the resulting security policy, since the majority of filesystem rules can simply be removed.

4.2 Network Policy

BPFCONTAIN confines network traffic at the socket level. *Network rules* grant access to various socket operations on the IPv4 and IPv6 socket families. Socket operations are grouped by use case and partitioned into `client`, `server`, `send`, and `receive` categories accordingly. These categories may be mixed and matched according to the required level of access. An access level of `client` grants the ability to connect to a network socket, while an access level of `server` allows a container to create, bind, and shutdown a socket as well as listen for and accept connections. Similarly, `send` access grants the ability to send data (or write) to a network socket, while `receive` access grants the ability to receive data (or read) from a network socket.

Since Unix domain sockets are used for interprocess communication rather than network communication, they are handled separately by IPC policy (c.f. Section 4.2.1). A container typically has no need for other address families beyond basic networking and inter-process communication. Therefore, in our current prototype, all other address families are implicitly denied, although this may be subject to change in future iterations.

4.2.1 IPC Policy

BPFCONTAIN enforces inter-process communication (IPC) policy at the granularity of its representation of containers. A given container can *always* perform IPC between its *own* processes. To enable IPC across containers, *both* container security policies must specify an *ipc rule* which explicitly allowlists the other policy by specifying its name. For instance, to allow IPC between *policy A* and *policy B*, *policy A* would require an `ipc: B` rule and *policy B* would require an `ipc: A` rule. At runtime, BPFCONTAIN enforces checks to make sure that both containers belong to the same IPC namespace.

BPFCONTAIN enforces its IPC policy on all inter-process communication facilities supported in the LSM infrastructure, including Unix domain sockets, signals, and SystemV IPC and shared memory.

4.3 Capability Policy

Over-provisioning and abuse of POSIX capabilities is endemic in extant container implementations [11, 49]. To rectify the problem of overprivilege, BPFCONTAIN strictly limits the use of POSIX capabilities by all containers, including those which have been marked default-allow. The only way to allow the use of a given POSIX capability in a BPFCONTAIN container is to explicitly allow it by adding a *capability rule* with the corresponding capability.

Note that allowing the use of these capabilities is *not* the same as granting additional capabilities; a process must still possess the corresponding POSIX capability in order to use it. For example, in order for a process to use the `CAP_SYS_ADMIN` privilege, it must both already have this privilege and be running under a policy with an explicit `CAP_SYS_ADMIN` capability rule. In this way, BPFCONTAIN's capability policy can be thought of as a mask over the set of all possible capabilities.

4.4 Implicit Policy

Listing 4.3: Sample BPFCONTAIN policy for a container that runs Apache `httpd` together with `mysqld`. In practice, this policy could be further simplified by relying on BPFCONTAIN's implicit overlayfs policy.

```
1  name: my_webapp
2  entry: >
3    mysqld $(SQL_ARGS) &
4    httpd $(HTTPD_ARGS)
5  allow:
6    - file: /run/apache2.pid, rwd
7    - subdir: /run, rc
8    - subdir: /var/www, r
9    - subdir: /var/www/cgi, rxm
10   - subdir: /var/cache/apache2, rwc
11   - subdir: /var/lib/mysql, rwc
12   - subdir: /var/lib, rm
13   - subdir: /var/log/apache2, rac
14   - subdir: /var/log/mysql, rac
15   # Network access
16   - net: server, send, recv
17   # Bind to privileged ports
18   - capability: netBindService
19   # Inter-process communication between mysqld
20   # and httpd is allowed implicitly, as both
21   # processes exist in the same container
```

In addition to its explicit security policy, defined in policy files, BPFCONTAIN enforces a number of *implicit policies* on all containers, which are enforced regardless of configuration. In general, these policies restrict access to operations that no sane container should ever require. In total, BPFCONTAIN enforces implicit policy on 11 LSM hooks, denying access to the `bpf(2)` system call, the kernel's keyring facilities, the ptrace system call, filesystem mounts, and all of the sensitive operations gated by the kernel's lockdown LSM [28].

Besides the implicit policy enforced via LSM hooks, BPF-

CONTAIN takes advantage of eBPF to instrument other areas of the kernel which are not directly covered by LSM hooks. In this way, BPFCONTAIN can dynamically harden the kernel against a variety of additional container-specific attacks such as namespace escapes and host privilege escalation attacks [25]. We cover the specific details of this extra enforcement in Section 5.2.

The key advantage to these implicit policies, however, is that they make container-level policies remarkably simple. Listing 4.3 shows a sample policy for a container running both apache and mysql. Notice that the bulk of the policy is taken up by filesystem rules, and even then the policy is relatively small, as standard activity within the container is allowed. If we had configured an overlay filesystem (as is done by standard container runtimes) this policy could be even simpler, requiring no filesystem permissions at all. (See Section 8.1.)

5 Implementation

BPFCONTAIN consists of three functional components: (1) a privileged daemon, running in userspace; (2) a minimal shim application for launching containers; and (3) eBPF programs attached to various system events and LSM hooks. Figure 5.1 provides a detailed overview of BPFCONTAIN's architecture. In the rest of this section, we discuss the userspace and kernelspace components respectively.

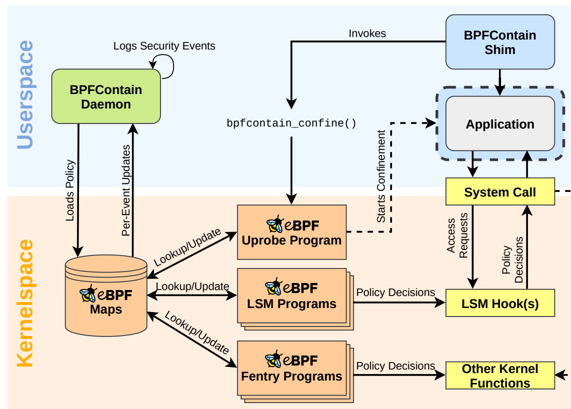

Figure 5.1: A detailed depiction of BPFCONTAIN's architecture. The daemon is responsible for loading and managing BPFCONTAIN's eBPF programs and maps, translating and loading policy into the kernel, and logging security events to userspace. eBPF programs attach to LSM hooks and other security-critical kernel functions to enforce per-container policy. A userspace shim application is responsible for launching the containerized application and starting confinement. Unlike other container implementations, the shim has no need to directly communicate with the daemon and runs with no additional privileges.

5.1 Userspace Components

In userspace, BPFCONTAIN consists of a privileged daemon and an unprivileged shim application used to launch containers. The daemon is responsible for loading and managing BPFCONTAIN's eBPF maps and programs, loading per-container policy into the kernel, and logging security events for provenance. The shim's only purpose is to invoke the `bpfcontain_confine` library call to initiate confinement followed by an `execve(2)` call to launch the correct entrypoint application.

Unlike traditional container implementations, the shim (and consequently any container launched using the shim) requires no special privileges. To obviate the need for communicating with the daemon directly, we instead instrument a uprobe (userspace probe) eBPF program on the `bpfcontain_confine` library call. Whenever a userspace application makes this library call, it traps to the uprobe which then creates a new container for the process and associates it with the correct security policy. From the application's perspective, this procedure is totally transparent.

Both the daemon and shim are written in Rust, a fast and reliable systems programming language with strong memory- and thread-safety guarantees. BPFCONTAIN leverages libbpf-rs and BPF CO-RE (compile once, run everywhere) to load its eBPF programs and maps into the kernel. With CO-RE, BPFCONTAIN embeds its eBPF object code directly into the resulting executable, allowing for a single binary to be compiled once and executed on any system running the minimum required kernel version. We expect that this property will prove highly advantageous for cloud-based and IoT-based deployments.

5.2 Kernelspace Components

On the kernel side BPFCONTAIN leverages a variety of eBPF programs and maps to enforce policy, log security events, and track the state of running containers. In total, we instrument 38 eBPF LSM programs, two fentry (kernel function entry) programs, two scheduler tracepoints, and one uprobe (userspace probe). Taken together, these components provide a complete abstraction for containers and container behaviour in kernelspace and offer an effective mechanism for enforcing least-privilege on running containers.

eBPF Maps. To store policy, track the state of running containers, and log security events to userspace, BPFCONTAIN employs several eBPF maps. These maps are specialized data structures which reside in the kernel and provide bi-directional (by default) lookup and update capabilities to privileged userspace applications and eBPF programs.

A processes map tracks the state of containerized processes and manages their associations with running containers. Similarly a containers map tracks the state of running containers, including policy association, information about namespace

and cgroup membership, whether the container is tainted, and a reference count of how many processes are running under the container.

We additionally define one map per policy category (e.g. file, filesystem, and network policy) and mark each policy map read-only after populating it from userspace. When a container requests access to a sensitive resource, eBPF programs query the policy maps and the state of the running container and use this information to come to a policy decision. A final specialized map acts as a ring buffer to store and forward security events (e.g. policy denials) to userspace so that the BPFCONTAIN daemon can log them.

LSM Programs. The majority of BPFCONTAIN's policy enforcement mechanism is implemented using eBPF programs attached to LSM hooks. These LSM hooks are strategically positioned in various security-critical sections of kernel code and define the canonical interface for implementing custom access control policy in the kernel. These LSM programs can coexist with any other Linux security module running on the system (e.g. AppArmor, SELinux, or Yama) and work co-operatively with other implementations to come to a final policy decision. This means that, although BPFCONTAIN is designed to completely replace existing LSMs for container security, exclusivity is not a requirement.

Hot Patching Kernel Vulnerabilities. While LSM programs provide a strong basis for policy enforcement, we found that the kernel could benefit from additional hardening in areas which are not directly protected by LSM hooks. For instance, Xin *et al.* [25] identified a common class of container privilege escalation attacks which work 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 elevated 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 [25]. While effective, this solution is inflexible in that it assumes a specific container abstraction based on namespace membership and requires an out-of-tree kernel patch.

In BPFCONTAIN, we implement a similar mitigation technique but instead use a special eBPF program typed called an fentry probe. Fentry probes replace kernel function entrypoints with a shim function that trampolines to an eBPF program. BPFCONTAIN attaches such an fentry probe to the `commit_creds` function and uses it to check for privilege escalation within a container. If the probe detects privilege escalation, it simply kills the offending process.

We apply a similar technique to the `switch_task_namespaces` function to prevent a container from escaping namespace isolation. In principle, it is possible to add arbitrarily many such fentry programs to BPFCONTAIN as new kernel-level vulnerabilities are discovered. These "hot patches" can then be applied dynamically, simply by reloading the BPFCONTAIN daemon.

Scheduler Tracepoints. To keep track of process state and container membership, we instrument tracepoint eBPF programs on the scheduler, tracking task creation and task exits. When BPFCONTAIN detects that a task has been created, it checks to see if its parent is in a BPFCONTAIN container. If this check passes, the child process is associated with the same container and we atomically increment the container's reference count. In this way, BPFCONTAIN recursively builds its own per-container process tree.

Similarly, when a task exits, we check if it belongs to a container and decrement the corresponding reference count. Once a container's reference count reaches zero, BPFCONTAIN removes it from the containers map, freeing up space for a new container to take its place.

Extending the Kernel ABI with Uprobes. An important difference between BPFCONTAIN and traditional container implementations is that a process can containerize itself without requiring any additional privileges. We accomplish this via a shim application that makes a single library call, `bpfcontain_confine`, before calling `execve(2)` to launch the target application. On the kernel side, `bpcontain_confine`'s logic is implemented using a uprobe (userspace probe), which replaces the function address with a trap to an eBPF program. This eBPF program first checks to ensure that the calling process is not already associated with a container, to prevent an exiting container from escaping confinement. If this check succeeds, BPFCONTAIN creates a new container, associates it with the correct policy, adds the current process to the container, and passes control back to userspace.

6 Evaluation

Here we evaluate the security and performance of BPFCONTAIN. We evaluate its security by analyzing its implementation in the context of the threat model presented in Section 3.1, while performance is empirically analyzed relative to programs running with no confinement and as confined by AppArmor. While neither of these evaluations are exhaustive, they show that BPFCONTAIN is strong in the context of its threat model while imposing comparable overhead to other confinement solutions.

6.1 Security

To prevent attacker-controlled containers from making system calls that could potentially compromise other containers or the host system, we must ensure that BPFCONTAIN acts as

a reference monitor with respect to kernel operations. According the Anderson's reference monitor model [2], a secure reference validation mechanism must satisfy the properties of *complete mediation*, *tamper resistance*, and *verifiability*. BPFCONTAIN's containment guarantees are proportional to the degree these three properties are satisfied. Below we discuss the degree to which they hold for BPFCONTAIN.

Complete Mediation. Recall that most of BPFCONTAIN's policy enforcement mechanism leverages the Linux Security Modules framework exposed by the kernel. If we assume that the property of complete mediation holds for the LSM framework itself, 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. Where feasible, BPFCONTAIN uses its own mechanisms to prevent tampering with its state while it is running. This protection involves multiple mechanisms, described below.

BPFCONTAIN is potentially vulnerable to rogue `bpf(2)` system calls to modify or remove its code and maps. The kernel gates access to the `bpf(2)` system call with the CAP_SYS_ADMIN capability, which means a process would effectively require root privileges before accessing any map on the system [26]. To protect against compromised privileged processes, BPFCONTAIN includes specific logic in its LSM probe on the `bpf(2)` system call itself, preventing any userspace process other than 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 [26].

BPFCONTAIN should continue to enforce protections even if its userspace daemon is terminated. Normally eBPF code is loaded as long as the associated file descriptor is open [47]; thus, when file descriptors are closed on program termination, the associated eBPF code and maps are freed. BPFCONTAIN ensures their persistence by pinning all of its maps and programs to a special filesystem called `bpffs`, thus incrementing the reference counts on the file descriptors. BPFCONTAIN also instruments an LSM probe preventing any other process from calling `unlink(2)` on its pinned file descriptors.

BPFCONTAIN is still vulnerable to kernel-level compromises, for example through code injection attacks. This vulnerability is no more than that of any other Linux kernel security mechanism, however. Further, as explained above, BPFCONTAIN has significant protections against unauthorized yet privileged users and processes. While its implementation is not quite as locked down as SELinux or AppArmor, its current implementation is hardened without a significant usability impact.

Verifiability. While BPFCONTAIN has not been formally verified, its design and implementation do facilitate verification in multiple ways. First, BPFCONTAIN's eBPF code is run through the eBPF bytecode verifier whenever it is run. One of the biggest challenges in working with eBPF is the strictness of the verifier as it imposes so many restrictions in order to ensure safety. These very limitations, however, prevent many typical coding errors and form a solid basis for more advanced verification approaches.

BPFCONTAIN has a comparatively small kernel-level codebase (well under 2000 lines of kernelspace code) compared to conventional LSM implementations such as SELinux or AppArmor, further facilitating manual audits or automated verification. However, given that BPFCONTAIN is ultimately only as secure as the rest of the Linux kernel, such verification can only provide modest additional security guarantees.

6.2 Performance

To establish the performance overhead of BPFCONTAIN as a drop-in replacement for confinement solutions such as AppArmor, we evaluated its performance using several benchmarking tests provided in the Phoronix Test Suite [22], to our knowledge the most comprehensive Linux benchmarking platform. Table 6.1 describes each benchmarking test in detail. To establish a baseline for comparison with AppArmor, we ran each test under five distinct configurations, described in Table 6.2. See Appendix Table A.1 for full results of each benchmark.

Tests were run in a KVM-enabled QEMU virtual machine running a 5.10 Linux kernel on an idle host machine. The virtual machine was given 8 virtual CPUs at 2.99GHz and 16GB of RAM.

Figure 6.1 presents the comparative percent overheads of all benchmarking tests across each system configuration, as compared to the configuration without any security enabled. Our results indicate that there is no significant difference between the overhead imposed by BPFCONTAIN and AppArmor, as all percent differences were well within the margin of error (c.f. Table A.1). Thus we conclude that BPFCONTAIN is competitive with AppArmor in terms of performance overhead on both confined and unconfined processes. Further, in all cases BPFCONTAIN exhibits less than 16% overhead on the running system, which we find to be acceptable in practice.

Table 6.1: A description of the benchmarking tests.

Test	Description
Apache Webserver	Measures requests per second of serving static HTML content via the Apache `httpd` webserver.
Kernel Compilation	Measures time taken to build the Linux kernel.
Create Files	Measures time taken to create, write, and delete files.
Create Threads	Measures time taken to create new threads.
Create Processes	Measures time taken to fork into new processes.
Launch Programs	Measures time taken for fork and execute a dummy program.
Memory Allocations	Measures time taken allocate and free small chunks of memory (4 – 128 bytes).

Table 6.2: A description of the system configurations during benchmarking.

Configuration	Description
No Security	No LSM is running on the system.
AppArmor Base	AppArmor is running without any security profiles enabled.
AppArmor Allow	AppArmor is running and enforcing a security profile that allows all operations.
BPFCONTAIN Base	BPFCONTAIN is running without any security profiles enabled.
BPFCONTAIN Allow	BPFCONTAIN is running and enforcing a security profile that allows all operations. This exercises the full code path of all BPF programs.

7 Related Work

Several researchers [25, 34, 49] have examined various aspects of the container security spanning various container management platforms and confinement mechanisms. Sultan *et al.* [49] examined the container security landscape in detail, identifying strengths, weaknesses, patterns in academic

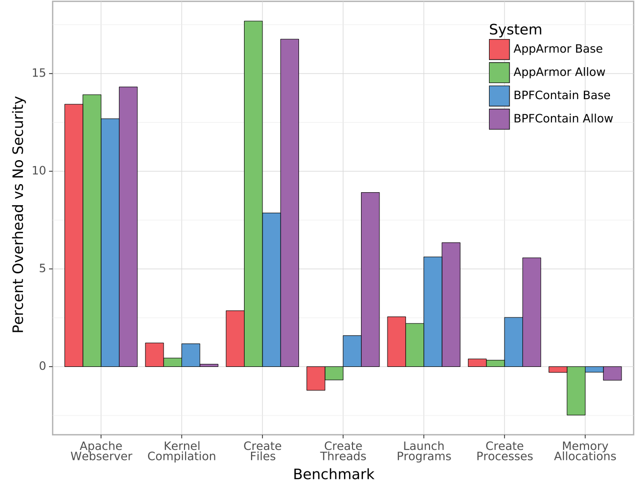

Figure 6.1: Percent overhead of various configurations vs no security, lower percentages are better. Negative results were found to be within margin of error.

literature, and promising future work opportunities. Their recommendations included work towards a container-specific LSM [49], which inspired the research direction for BPFCONTAIN. Xin *et al.* [25] presented a detailed taxonomy of container security exploits and analyzed container management platforms' security against their exploit database. Mullinix *et al.* [34] 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 [4, 21, 43] has proved a lucrative technique for identifying areas of weakness in container configurations. Shu *et al.* [43] 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 [43]. Kwon and Lee [21] 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.* [4] 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.* [6] 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.* [23] 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 [23]. Xin *et al.* [25] 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.* [32] 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.* [50] 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.* [49] 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 [39, 40] 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 [9]. Thus, Landlock has not been merged into the mainline kernel and will likely remain out-of-tree going forward. Unlike Landlock, Singh's KRSI [9, 44] 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.* [13] introduced bpfbox as the first full process confinement mechanism using these eBPF LSM hooks. BPFCONTAIN differs from bpfbox in three key ways: 1) BPFCONTAIN confines containers (sets of processes and associated resources) while bpfbox confines individual processes; this also affords a significant simplification of security policy, since implicit rules can be enforced at the per-container granularity 2) BPFCONTAIN is implemented using BPF CO-RE and Rust rather than bcc and Python, greatly reducing it storage and runtime overhead, and 3) BPFCONTAIN moves beyond just LSM-layer enforcement to apply additional kernel hardening against privilege escalation attacks that can undermine its protection.

8 Discussion

When designing security solutions, we are often at the mercy of past design decisions. One of the core insights of computer security is that it is easier to build security in rather than add it afterwards. Central to this idea is that many security problems arise at the architectural level, and it is hard to change the architectures of deployed systems. As a result, we often find ourselves integrating security mechanisms in fundamentally non-optimal contexts, simply because that is where they can be implemented.

BPFCONTAIN is a demonstration of how operating system extensibility can enable the development of new security abstractions that more closely match the problems at hand. Domain-specific policy language no longer has to be implemented using existing (general-purpose) security mechanisms; instead, we can implement security policies using the fundamental functional abstractions of the operating system.

8.1 Future Directions

Integration with Existing Container Runtimes. Our focus with BPFCONTAIN has been on isolation rather than virtualization. A potential next step would be to integrate BPFCONTAIN with Docker [11], Kubernetes [20], or OpenShift [37]. Given that these container runtimes are largely Open Container Initiative (OCI) compliant [30], it should be possible to integrate with them in a portable fashion. Further, since eBPF can transparently instrument userspace code, such integration would in principle be possible without making changes to the source code of the underlying container runtime. Policy generation and enforcement could be tied directly with container configuration, offering streamlined, precise, yet simple confinement specifications that fail closed rather than fail open.

Rootless Containers in eBPF. Another potential path would be to add virtualization and management capabilities directly to BPFCONTAIN. The advantage of doing so would potentially be greatly simplified userspace tools due to enhanced kernel-level functionality implemented through new eBPF helpers. 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 policies as well. Not only would such an extension enable fully application-transparent namespace and cgroup management, but it would also obviate the need for the root privileges

currently required by container management systems.

On the policy language side, the integration of virtualization with BPFCONTAIN's enforcement mechanisms 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 reduce complexity in filesystem and file rules. For example, one mount rule could replace a series of file rules specifying access to required shared libraries. Thus, by further refining the abstractions presented to userspace by the kernel, we can enable solutions that are simpler, easier to configure, and potentially much more secure.

Running BPFCONTAIN without the Daemon After the BPFCONTAIN daemon loads its eBPF programs and maps into the kernel, its only remaining purposes is to log security events to userspace. With the ability to pin eBPF objects (thus maintaining a reference count), it would in principle be possible to completely remove the need for the daemon altogether. Naturally, this would significantly reduce BPFCONTAIN's attack surface, and this would mark an obvious improvement over existing container runtimes like Docker and Kubernetes which rely on privileged daemons in order function correctly. While, in the current implementation, disabling the BPFCONTAIN daemon would disable its audit logging capabilities, it may be possible to make use of a new eBPF iterator program type added in recent kernels to implement similar functionality.

8.2 Prototype Limitations

While we believe BPFCONTAIN is a promising approach to container confinement, in its current form it has some limitations. One is that there are hard limits on policy complexity due to the current limits of eBPF. eBPF's maps must be constrained to some fixed size that is determined at map creation time. Since BPFCONTAIN uses eBPF maps to store per-container policy, the maximum map size bounds the number of possible rules for each policy category. This upper bound is not strictly an issue since BPFCONTAIN loads 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, which would require handling this map size restriction at runtime. Given eBPF's strong safety guarantees are due in part to draconian restrictions on dynamic memory allocation, removing this limitation is non-trivial. Work on garbage collected map types, however, may solve this issue in the near future.

9 Conclusion

This paper presented BPFCONTAIN, a novel least-privilege implementation for container security. BPFCONTAIN exposes a simple YAML-based policy configuration language to userspace that conforms to existing container management mechanisms' semantics while supporting ad-hoc confinement use cases through high-level policy rules and optional default-allow enforcement. Because BPFCONTAIN is written in eBPF, it can be deployed on virtually any system running a recent Linux kernel with no kernel-level changes and none of the risks of out-of-tree kernel modules. eBPF also allows BPFCONTAIN to protect itself against privilege escalation exploits that could prevent its functioning.

When integrated with OCI-compliant container management systems, BPFCONTAIN will provide strong yet flexible container confinement, enabling more secure multi-tenant container deployments. It also helps demonstrate the potential of using eBPF to extend the Linux kernel with new security abstractions.

References

[1] Marcelo Amaral, Jorda Polo, David Carrera, Iqbal Mohomed, Merve Unuvar, and Malgorzata Steinder, "Performance evaluation of microservices architectures using containers," in *2015 IEEE 14th International Symposium on Network Computing and Applications*, IEEE, 2015, pp. 27–34.

[2] 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.

[3] 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).

[4] 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.

[5] Neil Brown, "Control groups, part 1: On the history of process grouping," *LWN.net*, Jul. 2014. [Online]. Available: https://lwn.net/Articles/603762/.

[6] 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.

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

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

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

[10] 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.

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

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

[13] 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.

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

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

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

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

[18] Michael Kerrisk, "Namespaces in operation, part 1: Namespaces overview," *LWN.net*, Jan. 2013. [Online]. Available: https://lwn.net/Articles/531114/.

[19] 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.

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

[21] 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.

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

[23] 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.

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

[25] 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.

[26] 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).

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

[28] 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).

[29] 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).

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

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

[32] 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.

[33] 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.

[34] 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.

[35] Onur Mutlu and Jeremie S Kim, "Rowhammer: A retrospective," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 8, pp. 1555–1571, 2019.

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

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

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

[39] 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).

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

[41] 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.

[42] 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.

[43] 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.

[44] 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/.

[45] 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.

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

[47] 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).

[48] 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.

[49] 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.

[50] 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.

[51] 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.

A Appendix A: Benchmarking Results

Table A.1 lists detailed benchmark results comparing BPFContain to AppArmor.

Table A.1: Full benchmarking results comparing AppArmor and BPFCONTAIN.

Test	System	Units	Mean	Stdev	Percent Overhead
Apache Webserver	Base	req/sec	23258.19	108.03	—
Apache Webserver	BPFContain Base	req/sec	20306.59	326.57	12.69%
Apache Webserver	BPFContain Allow	req/sec	19928.84	203.66	14.31%
Apache Webserver	AppArmor Base	req/sec	20134.33	534.40	13.43%
Apache Webserver	AppArmor Allow	req/sec	20020.81	244.99	13.92%
Kernel Compilation	Base	sec	188.88	1.73	—
Kernel Compilation	BPFContain Base	sec	191.10	1.48	1.17%
Kernel Compilation	BPFContain Allow	sec	189.12	2.08	0.13%
Kernel Compilation	AppArmor Base	sec	191.18	1.59	1.21%
Kernel Compilation	AppArmor Allow	sec	189.72	1.53	0.44%
Create Files	Base	usec	19.32	0.14	—
Create Files	AppArmor Base	usec	19.87	0.18	2.86%
Create Files	AppArmor Allow	usec	22.74	0.21	17.69%
Create Files	BPFContain Allow	usec	22.56	0.15	16.76%
Create Files	BPFContain Base	usec	20.84	0.22	7.87%
Create Threads	Base	usec	21.48	1.74	—
Create Threads	AppArmor Base	usec	21.22	1.31	-1.21%
Create Threads	AppArmor Allow	usec	21.33	1.30	-0.68%
Create Threads	BPFContain Allow	usec	23.39	1.20	8.91%
Create Threads	BPFContain Base	usec	21.82	1.48	1.59%
Launch Programs	Base	usec	61.35	0.55	—
Launch Programs	AppArmor Base	usec	62.92	0.66	2.55%
Launch Programs	AppArmor Allow	usec	62.71	0.71	2.21%
Launch Programs	BPFContain Allow	usec	65.24	0.58	6.35%
Launch Programs	BPFContain Base	usec	64.80	0.47	5.62%
Create Processes	Base	usec	41.33	2.28	—
Create Processes	AppArmor Base	usec	41.49	1.28	0.40%
Create Processes	AppArmor Allow	usec	41.47	1.89	0.33%
Create Processes	BPFContain Allow	usec	43.63	1.82	5.57%
Create Processes	BPFContain Base	usec	42.37	2.29	2.52%
Memory Allocations	Base	nsec	93.80	0.40	—
Memory Allocations	AppArmor Base	nsec	93.53	0.39	-0.30%
Memory Allocations	AppArmor Allow	nsec	91.48	0.28	-2.48%
Memory Allocations	BPFContain Allow	nsec	93.15	0.62	-0.70%
Memory Allocations	BPFContain Base	nsec	93.54	0.29	-0.29%