bpfbox: Simple Precise Process Confinement with eBPF

William Findlay
Carleton University
williamfindlay@cmail.carleton.ca

Anil Somayaji
Carleton University
soma@scs.carleton.ca

David Barrera
Carleton University
david.barrera@carleton.ca

ABSTRACT

Process confinement is a key requirement for workloads in the cloud and in other contexts. Existing process confinement mechanisms on Linux, however, are complex and inflexible because they are implemented using a combination of primitive abstractions (e.g., namespaces, cgroups) and complex security mechanisms (e.g., SELinux, AppArmor) that were designed for purposes beyond basic process confinement. We argue that simple, efficient, and flexible confinement can be better implemented today using eBPF, an emerging technology for safely extending the Linux kernel. We present a proof-of-concept confinement application, `bpfbox`, that uses less than 2000 lines of kernelspace code and allows for confinement at the userspace function, system call, LSM hook, and kernelspace function boundaries—something that no existing process confinement mechanism can do. Further, it does so using a policy language simple enough to use for ad-hoc confinement purposes. This paper presents the motivation, design, implementation, and benchmarks of `bpfbox`, including a sample web server confinement policy.

CCS CONCEPTS

• **Security and privacy → Operating systems security**; **Access control**.

KEYWORDS

eBPF; Sandboxing; Operating System Security; Application Confinement; Access Control; Linux

ACM Reference Format:
William Findlay, Anil Somayaji, and David Barrera. 2020. `bpfbox`: Simple Precise Process Confinement with eBPF. In *2020 Cloud Computing Security Workshop (CCSW'20), November 9, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3411495.3421358

1 INTRODUCTION

Process confinement is a critical problem for cloud computing. Whether one is running conventional servers, container-based workloads, or untrusted third-party code, a fundamental requirement is that some processes need to be restricted to enforce least privilege. Today, we have a wide variety of technologies on Linux for confining processes: user and group permissions, Linux capabilities, SELinux [45], AppArmor [12], `seccomp(2)` [41], cgroups [9], namespaces [26] and even the venerable `ptrace(2)` [36]. While these technologies can be used to contain processes, none were designed specifically for this task. As a result, process confinement implementations must combine these disparate approaches in order to create working solutions.

When security mechanisms require complex implementations, we can expect many vulnerabilities to be found, as they have for every major Linux confinement implementation[1]. But even worse, complexity makes changes more difficult, stifling innovation. The natural solution for this sort of challenge would be to implement an extension API which would allow for different process confinement solutions. The Linux community long ago recognized the need to support different security abstractions, resulting in the LSM (Linux Security Modules) framework [33]. LSM provides the interface used by mandatory access control systems such as AppArmor and SELinux. Developing new process containment abstractions on top of LSM and other in-kernel APIs is certainly possible; however, kernel modules can be very expensive to develop and maintain, and small errors in them can lead to catastrophic failures. Due to stability and security concerns, many environments will refuse to use solutions that require their own kernel modules. These challenges are why current confinement solutions build on top of existing kernel functionality rather than adding their own, despite the inevitable implementation complexity.

The key insight of this paper is that recent additions to the Linux kernel have enabled a third path to implementing process confinement: Extended Berkeley Packet Filter (eBPF) [47]. Originally developed to improve packet filtering, in Linux BPF has been extended to allow fine-grained introspection of kernel and userspace behavior. As with kernel modules, code is loaded by the root user into the kernel; unlike kernel modules, eBPF is a bytecode language that is verified at load time and then just-in-time compiled before being linked into the kernel. This verification step, combined with the limitations of eBPF bytecode, allows kernel functionality to be extended in a safe yet performant way.

Here we present `bpfbox`, a prototype process confinement system consisting of a policy language and eBPF-based implementation that can run on Linux 5.8 and newer kernels. The key advantages of `bpfbox` are that it combines simple yet flexible policies with an implementation that can be easily changed and improved, exploiting the flexibility of eBPF. `bpfbox` is implemented in bcc [2], a development toolchain for Linux eBPF programs that combines Python userspace code with eBPF code written in a restricted subset of C. While policy errors can impede the functioning of specific programs, implementation errors in `bpfbox` will not harm the system, due to the safety guarantees of eBPF.

`bpfbox` policies advance the state of the art in process confinement by allowing simpler yet more precise policies to be written. Specifically, it is possible to go beyond system calls and network

CCSW'20, November 9, 2020, Virtual Event, USA
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8084-3/20/11.
https://doi.org/10.1145/3411495.3421358

[1]A small selection of CVEs show how common confinement vulnerabilities are. seccomp: CVE-2019-2054, AppArmor: CVE-2019-18814, SELinux: CVE-2020-10751, Docker: CVE-2020-13401, snap: CVE-2019-11503, flatpak: CVE-2019-8308

connections and instead specify restrictions on userspace and kernelspace functions. Instead of simply preventing a process from making a call to `execve(2)` (or an `execve` of specific program binaries), `bpfbox` can specify that `execve` calls for a process must originate in a specific function and should optionally pass certain functional checks in the kernel (e.g., that the program being run is an ELF binary rather than a script). While such additional instrumentation does result in additional runtime overhead, as we show the overhead is quite modest in practice—for example, when alerting on *every* security operation, `bpfbox` slows down a web server by around 8.4%, slightly less than running AppArmor in a similar configuration. Further, `bpfbox` is implemented using less than 2000 lines of in-kernel code, smaller than seccomp-bpf and much smaller than SELinux. The relative simplicity of `bpfbox` means that it can be used on its own to do ad-hoc process confinement; its flexibility, however, means that it could be used to simplify and extend the confinement technology in container management systems or even in regular Linux distributions. Looking forward, we see `bpfbox` as a proof of concept that demonstrates the potential of eBPF to enable more secure fine-grained resource sharing in the cloud.

The rest of this paper proceeds as follows. We first review standard Linux security mechanisms in Section 2. Next, we describe the rationale for `bpfbox` (Section 3) in the context of existing process confinement solutions. We explain eBPF and related technologies in Section 4. In Section 5, we explain how `bpfbox` policies are enforced using eBPF, and we describe the `bpfbox` policy language in Section 6. We present a web server policy example in Section 7. Section 8 presents a quantitative evaluation of `bpfbox`'s performance impact. Section 9 discusses limitations, potential enhancements to the current research prototype, and future research directions. Section 10 concludes.

2 LINUX CONFINEMENT MECHANISMS

Process confinement is extensively used on desktop, mobile, and server/cloud systems to protect systems from running partially or fully untrusted code. On the desktop, web browsers isolate processes running JavaScript loaded by web applications. On mobile devices, third-party applications are confined so they have limited access to system resources and other applications. On servers, containers separate applications from each other, enabling smooth deployment and load balancing. Many mechanisms are used to implement confinement on Linux systems. In this section we discuss these confinement mechanisms; in the following section we examine why these are insufficient.

On Linux, the main implementation technologies for containing processes are Unix discretionary access controls, mandatory access controls, cgroups, namespaces, Linux capabilities, and `seccomp`. We discuss each in turn below.

Unix DAC: The most basic way to confine processes in Linux is to make use of the traditional Unix user-oriented discretionary access controls that determine how a process's user and group IDs restrict what files, system calls, and other processes it can access. Unprivileged users can only send signals to their own processes, and access to files can be limited using per-inode permission bits. Unix users, however, are given significant access to system resources, as they do have a significant amount of trust given to them—after all, they are authorized users of the system. Further, the root user can do almost anything to a system. Fine-grained process confinement thus typically requires additional mechanisms.

Linux MAC: Mandatory access control (MAC) limits the privileges of all users, including the root user, and thus serves as another set of mechanisms that can help confine processes. On Linux, kernel modules implementing MAC policies use the Linux Security Module (LSM) API [33] to hook into the security checks made throughout the kernel. Whether the implementation is SELinux [45], AppArmor [20], TOMOYO [21], or other, all MAC implementations limit the access processes have to kernel resources, including files and network sockets, based on attributes associated with the process or thread.

Namespaces and cgroups: On Linux, namespaces and cgroups allow system resources to be partitioned. Namespaces partition resources in terms of naming, giving a group of processes a private view of enumerable system resources such as process IDs, filesystems, network sockets, and user IDs. Cgroups limit non-enumerable resources such as memory, CPU, and I/O bandwidth. By limiting the resources that a process can name and how much of that resource it can use, namespaces and cgroups provide the base abstractions for operating system virtualization. Because OS virtualization is fundamentally a way to separate groups of processes from each other, these mechanisms can also be used for the simpler case of process confinement.

Linux capabilities: The root user in Unix systems traditionally has access to all resources; thus if an attacker gains control of a process running as root, all security controls are null and void. Linux capabilities (based on the withdrawn POSIX.1e standard[2]) subdivide root's privileges into individual capabilities which a privileged process can retain or keep individually. Alternately, program binaries can be set to have specific capabilities when `execve`'d, thus getting more privileges than normal, but not getting all privileges a setuid root program would get. Linux capabilities help minimize the privileges given to processes that manage access to resources, whether they be networking, filesystems, devices, or kernel modules.

Ptrace: The `ptrace(2)` system call is used to allow one process to monitor and control the execution of another process. Debugging tools such as gdb and strace are fundamentally based on `ptrace`. Because a process controlled by `ptrace` can have its execution arbitrarily monitored or changed, unprivileged users can only `ptrace` their own unprivileged processes. Additionally, modern Linux distributions enable the Yama [8] LSM by default, which restricts access to `ptrace` such that only parent processes may `ptrace` their children; tracing other unrelated process requires changing a kernel runtime option. While `ptrace` isn't normally used in production environments, it can be used to confine process behavior by introducing traps on restricted behavior.

Seccomp: While Linux capabilities allow for restrictions on privileged operations, they aren't sufficient to limit the system calls that unprivileged processes can normally make. `seccomp(2)` [41] is a system call that causes a process to enter a restricted state where it can only respond to signals, terminate, and read and write to open file descriptors. Other system calls cause forcible process

[2]POSIX 1003.1e Draft 17

termination. Seccomp-bpf [14] is an extension to `seccomp` that allows additional system call rules to be defined through classic BPF (not eBPF) filters. For example, with seccomp-bpf `open(2)` can be allowed, but only for specific files. Note the filtering is very flexible because it is implemented using BPF; this flexibility, however, comes at the price of complexity.

We explain more about classic BPF and eBPF in Section 4.

3 MOTIVATION

To understand the potential benefit of lightweight process confinement using eBPF, consider how existing systems separate processes in a cloud context. Cloud computing can be said to be (in part) an evolving platform for allocating computational resources so that machine-level boundaries are less and less significant—applications can span multiple hosts, and multiple applications can share specific hosts. Process confinement is a fundamental part of this evolving cloud computing story. Here we discuss how non-Linux systems have addressed process confinement and the limitations of existing approaches to process confinement on Linux.

The process confinement problem dates back half a century [28]. Dozens of tools and frameworks, some more practical than others, have been proposed to limit the impact of untrusted software on the rest of the system [43]. The isolation provided by containers is really a form of operating system virtualization, something that has a long history on Unix, with implementations including the standard `chroot(2)`, BSD jails [25] and Solaris Zones [38]. While OS virtualization partitions resources, on its own it does not really implement least privilege: processes can still have dangerous levels of access to system and network resources. Fine-grained access control can be used to complement OS virtualization, as shown with Capsicum [23, 48]. Also, system call filtering has been widely used to confine untrusted applications [17, 18, 24, 37, 39], both with and without OS virtualization.

Modern containers on Linux are simply instances of OS virtualized systems that are packaged for easy deployment and management. To a developer they appear almost as separate hosts, just without their own OS kernel and (often) some core system services. The actual separation of processes, however, is hardly more than the separation between processes on a regular Unix system. Of particular concern is that sufficiently privileged process inside of a container can "break out" and access any resources on the system, and such privileges can be obtained using any number of privilege escalation attacks. Container security thus relies on exactly the same mechanisms to separate processes that are used on any reasonably secured server.

We can see this pattern with the separation mechanisms used by Docker [13]. Docker uses Linux cgroups, namespaces, and filesystem mounts to implement OS virtualization. Other security mechanisms are employed to actually make sure processes remain separated and operate with least privilege. Linux capabilities are dropped so that root processes inside of a container cannot break out. Seccomp is used to restrict access to dangerous system calls, with the default policy restricting many system calls (such as those for managing kernel modules). AppArmor or SELinux are frequently used to further limit what resources container processes can access. Thus, while Docker provides easy-to-use mechanisms to developers and systems administrators, under the hood it must deal with the full complexity of securing a service on a Linux host.

Desktop container solutions are no better. For example, snap [46], Flatpak [16], Bubblewrap [5], and Firejail [15] also implement virtualized operating systems through cgroups, namespaces, and filesystem mounts. This OS virtualization is augmented with extensive integration with the host operating system's security mechanisms (i.e., Linux capabilities, `seccomp`, and MAC frameworks). While packaged in a friendly way, under the hood full Linux userlands must be secured for each installed application.

Operating system virtualization on Linux was originally developed to support the needs of shared hosting, in particular to allow multiple customers to administer their own web servers without the overhead of separate physical boxes or multiple kernels. With Docker and snaps, this same technology is used for application management. While some containers can be very complex, using multiple binaries, libraries, configuration files, and processes, others can be as simple as a single statically-linked binary running as a single process. In such contexts the elaborate container machinery is really only being used as a way to conveniently limit the access of a single binary.

Consider that all of the mechanisms used to secure containers, from cgroups to SELinux, are simply different mechanisms for telling the Linux kernel what resources processes can access. We use these multiple mechanisms because they are available, each having been built to solve specific problems from the past. If we started from scratch, we would certainly have developed simpler, more coherent mechanisms for isolating processes. `bpfbox` can thus be seen as an attempt to develop such a mechanism, made feasible by eBPF.

4 EXTENDING LINUX WITH EBPF

For this work, we wished to implement a new process confinement mechanism using eBPF, a new Linux technology for adding code to the kernel safely and efficiently. To understand the significance of eBPF, here we place it in the context of similar work in operating systems research and practice. We also explain the history of eBPF, present its high-level design, and discuss how it has been previously used for security applications.

In research on extensible operating systems, there has been extensive work on ways to safely extend their functionality. Adding code to an operating system is the perfect way to add abstractions and change system behavior; it is also the perfect way to make a system crash or silently corrupt its filesystems. Most work on extensible operating systems has been on microkernels such as Mach [1] and L4 [31], which facilitate extensibility by putting most of the OS into userspace processes; however, systems like SPIN [3] allow for code written in a safe language to be added to kernelspace. eBPF thus has similarities to SPIN, except it was designed to facilitate system introspection, not application performance.

The closest system and predecessor to eBPF is Solaris's DTrace [6, 19], which was created to allow for safe introspection of production systems by dynamically extending the Solaris kernel. The idea was that small scripts could be written in a constrained language that could be compiled, verified for safety, and then loaded into kernelspace. For licensing and technical reasons, DTrace has not been ported to Linux. Instead, the Linux community has ended up building a superset of DTrace's functionality in the form of eBPF.

Classic BPF has a rich history in Unix-like systems. BPF [32] first emerged as a packet filtering solution for BSD Unix in 1993. The key insight of classic BPF was its use of a register-based filter machine and buffering of results before returning them to userspace, resulting in significant performance increases over contemporary packet filtering solutions. Since then, it has been incorporated into many operating system kernels, notably Linux, OpenBSD, and FreeBSD, where it continued to be predominantly used for packet filtering. A notable exception to this is the seccomp-bpf extension under the Linux kernel, in which classic BPF programs may be written to augment `seccomp`, as we discuss in Section 2.

In 2014, Starovoitov and Borkmann [47] merged a major rewrite of the BPF engine into the Linux kernel. This new incarnation of BPF, dubbed Extended BPF (eBPF), added 10 new general purpose registers, a new instruction set with just-in-time (JIT) compilation to native instruction sets, access to allowed kernel helpers, a rich collection of specialized data structures, and new program types to enable tracing various aspects of system behavior. Since its inception, eBPF has seen rapid development with a myriad of new features in each kernel release[3]. `bpfbox`'s in-kernel enforcement engine leverages many of the new features provided by eBPF and requires no modification of existing kernel source code, meaning that any recent Linux kernel with support for eBPF LSM programs can run `bpfbox` without needing to be patched or recompiled.

eBPF programs themselves consist of a set of eBPF instructions that may be written by hand or optionally compiled from a restricted subset of a higher level language such as C. Many development toolchains such as bcc [2] and libbpf [30] exist to make this process easier. These eBPF instructions are then submitted to the kernel using the `bpf(2)` system call where they undergo a verification process before being loaded for JIT compilation. This verification process involves checking for program safety to ensure that an eBPF program cannot damage the running system. Since eBPF programs must be *verifiably safe*, they are fundamentally restricted in what they can do—eBPF is **not** Turing-complete. For instance, eBPF programs are not allowed to exceed 1 million instructions, lack support for advanced looping constructs, and have very limited support for handling strings. Pointer arithmetic is also disallowed unless it can be proved not to overflow or underflow the corresponding region of memory. These restrictions afford eBPF a distinct advantage over extending kernel behavior—modifying the source directly or adding kernel modules—particularly from a security perspective, as many traditional bugs such as buffer overflows and memory corruptions are outright prevented.

While nearly all system call filtering approaches have required OS-level modifications to implement security policy enforcement, seccomp-bpf can now act as a standard interface for these tools on Linux. MBOX [27] is one such sandbox that uses seccomp-bpf and redirects filesystem calls to a layered, app-specific filesystem. Notably MBOX does not require any changes to the host operating system. `bpfbox` uses eBPF instead of `seccomp` to sandbox processes beyond system call filtering, while retaining a small codebase, allowing fine-grained filtering, and avoiding the usual race conditions that impact system call filtering systems [17].

Probably the closest work to our own is the eBPF-based sandboxing system Landlock [40], an out-of-tree patchset for Linux that allows exposing LSM hooks to unprivileged userspace applications. Like `bpfbox`, Landlock attaches eBPF programs to LSM hooks to make access control decisions. Unlike `bpfbox`, Landlock envisions unprivileged processes specifying their own LSM restrictions using its C API. The key challenge faced by Landlock is that of securely exposing eBPF and LSM to unprivileged processes, something that the primary developer now believes is no longer possible in a world with CPU-level vulnerabilities [10]. In contrast, `bpfbox` runs with root privileges and assumes that policies have also been installed by a privileged user. Rather than requiring code-level additions, `bpfbox` separates policy generation from application development.

5 BPFBOX IMPLEMENTATION

In this section, we explain the implementation of `bpfbox`'s policy generation and enforcement mechanisms. First, we present an overview of the userspace and kernelspace components of `bpfbox` (Section 5.1). Next, we discuss how `bpfbox` generates policy in userspace and stores and enforces it in kernelspace (Section 5.2) and manage process state (Section 5.3). Finally, we discuss how `bpfbox` logs per-event audit data in userspace (Section 5.4). `bpfbox` is free and open source software, available under the GPLv2 license at https://github.com/willfindlay/bpfbox.

5.1 Architectural Overview

Figure 1 depicts `bpfbox`'s architecture, which consists of a userspace daemon `bpfboxd`, a collection of eBPF programs for tracing events and enforcing policy, and several maps (eBPF data structures accessible from both kernelspace and userspace) for storing information about active processes and loaded policy. `bpfboxd` parses policy files, compiles and loads eBPF code into the kernel, and interacts with running eBPF code using eBPF maps. `bpfboxd` is primarily written in Python; however, it links in a stub C library to facilitate interactions with eBPF code as explained below.

When eBPF code is loaded into the kernel, it is associated with a specific event so that when that event occurs, the registered code is run. eBPF code can be associated with arbitrary functions in kernelspace (kprobes) or userspace (uprobes); however, as functions can and do change, eBPF code can be associated with explicitly defined tracepoints in kernelspace or userspace (as USDTs). Through KRSI (Kernel Runtime Security Instrumentation) [10, 44], eBPF code can also be associated with any LSM hook (see Section 5.2).

While eBPF programs are loaded and eBPF maps can be accessed through the `bpf(2)` system call, USDT provides a way for userspace code to directly call eBPF code in the kernel. An eBPF program (loaded via the `bpf(2)` system call) defines kernelspace functions and associates them with USDTs defined in userspace stub functions. When those stub functions are called, they trap directly to the associated eBPF program. `bpfboxd`'s linked-in C library defines such USDT stub functions, allowing it to define a custom interface between its userspace and kernelspace portions, bypassing the normal system call interface.

[3]An up to date list of features: https://github.com/iovisor/bcc/blob/master/docs/kernel-versions.md

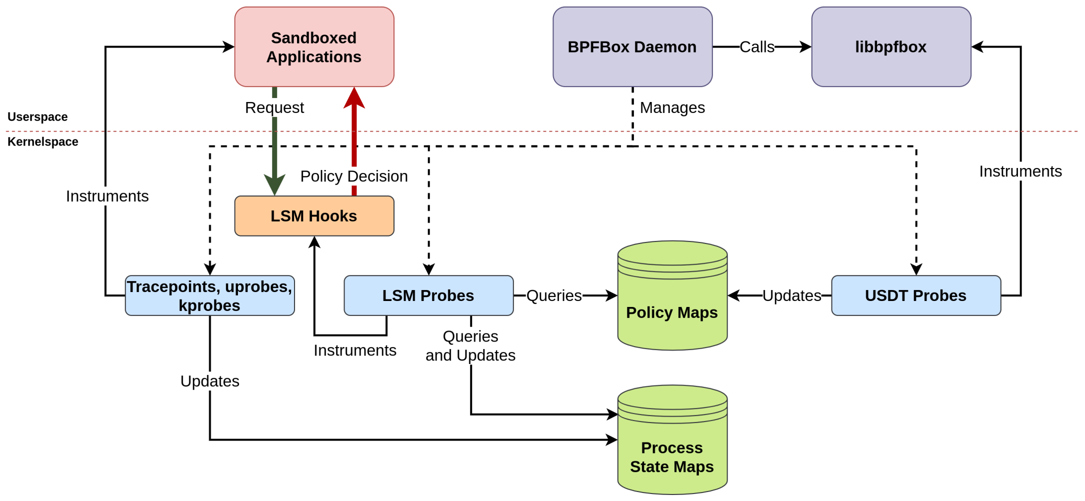

Figure 1: An overview of `bpfbox`'s components. `bpfbox` relies on three major categories of eBPF program: USDT probes which instrument `libbpfbox` to load and update policy; tracepoints, kprobes, and uprobes to instrument application behavior and manage process state; and LSM probes to enforce policy based on the information stored in policy and process state maps.

5.2 Implementing Policy

`bpfbox` policies are written using a custom policy language. `bpfbox`'s policy language allows for specific operations to be allowed, audited (logged), and/or tainted. All operations not so specified are denied. Tainting is similar in spirit to Perl's classic taint mode [22], however, rather than marking data, it marks the entire process. Tainting allows for more restrictive policies to be enforced once a process has engaged in specific unsafe operations, say by reading from a network socket. We present the design and syntax of the `bpfbox` policy language in Section 6; here we discuss the functionality it provides and how it is implemented.

`bpfbox` policies are per executable and are stored in an exclusively root-controlled directory (by default, `/var/lib/bpfbox/`), written in `bpfbox`'s policy language (Section 6). When an executable is loaded, `bpfbox` loads the corresponding policy file (if it exists) and translates it into a series of function calls to USDT stub functions. These function calls trigger the corresponding eBPF code, thus recording the policy in the policy maps as a set of policy structures. A policy structure consists of three distinct access vectors: one to define tainting operations, one to define allowed operations, and one to define audited operations.

In order to enforce policy, `bpfbox` leverages the KRSI patch by KP Singh [10, 44] which was upstreamed in Linux 5.7. This patch provides the necessary tools to implement MAC policies in eBPF by instrumenting probes on LSM hooks provided by the kernel. The eBPF program can then audit the event and optionally enforce policy by returning a negative error value. `bpfbox` instruments several LSM probes covering filesystem access, interprocess communication, network sockets, `ptrace(2)`, and even `bpf(2)` itself. When these hooks are called in the kernel, they trigger the execution of the associated eBPF program which is, in general, composed of the following six steps:

(1) Look up the current process state. If no state is found, the process is not being traced, so **grant access**.
(2) Determine the *policy key* by taking the executable's inode number and filesystem device number together as a struct.
(3) Look up the policy corresponding to the *policy key* calculated in step (2). If the process is *tainted* and no such policy exists, **deny access**.
(4) If the process is *not tainted* and the current access corresponds to a `TAINT` rule, **taint** the process and **grant access**.
(5) If the current access matches an `ALLOW` rule, **grant access**. Otherwise **deny access**.
(6) If the current access matches an `AUDIT` rule or access is **denied**, submit an *audit event* to userspace.

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

`bpfbox` can optionally augment the information provided by the LSM hooks themselves with additional context obtained by instrumenting other aspects of process behavior. For instance, profiles may optionally define *function contexts* which determine the validity of specified rules; a rule could specify that a certain filesystem access must occur within a call to the function `foo()` or that it must be audited within a call to the function `bar()`. The ability to combine various aspects of system behavior, both in kernelspace and in userspace, is a key advantage of an eBPF-based solution over traditional techniques. This allows for the creation of extremely fine-grained policies at the discretion of the policy author. The mechanisms by which this is accomplished are discussed further in Section 5.3.

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

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

Similarly to mandatory access control systems like SELinux [45] and AppArmor [12], `bpfbox` supports the ability to run in either *permissive mode* or *enforcing mode*. When running in permissive mode, `bpfbox` continues to audit denied operations, but allows them to continue unobstructed. This enables the user to debug policies before putting them into effect and also introduces the possibility of creating new policy based on the generated audit logs.

5.3 Managing Process State

In order for `bpfbox` to know what policy to apply to a given process, it must track the lifecycle of processes through the instrumentation of key events within the kernel. For this, `bpfbox` uses three tracepoints exposed by the scheduler: `sched:process_fork`, `sched:process_exec`, and `sched:process_exit`. Figure 2 shows the events that `bpfbox` instruments in order to track process state, along with their corresponding probe types. These tracepoints are used to create, update, and delete per-task entries in a global hashmap of *process states*. Each entry in the map is keyed by TID (thread ID). The entries themselves consist of a data structure that tracks policy key association and a 64-bit vector representing the *state* of the running process. This state vector is used to track whether the process is currently tainted and what important function calls are currently in progress.

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

The `sched:process_exec` tracepoint is triggered whenever a task calls `execve` to load a new program. `bpfbox` uses this tracepoint to manage the association of *policy keys* to a particular *process state*. `bpfbox` policy may optionally specify whether a transition from one profile to another may occur in a given call to `execve`; this transition is disallowed by default.

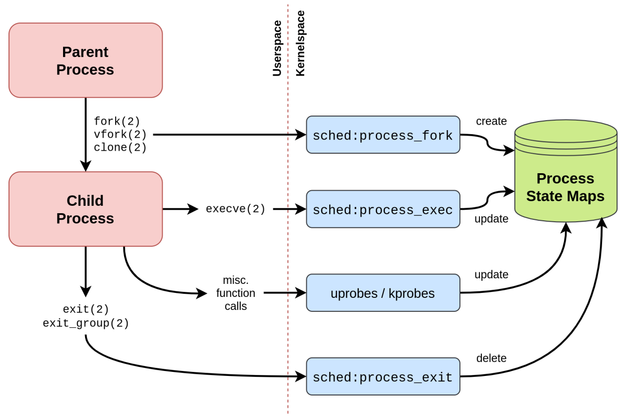

Figure 2: The various mechanisms that `bpfbox` uses to manage process state. Probes marked `sched:*` are tracepoints instrumenting scheduler events in the kernel. Uprobes and kprobes instrument userspace and kernelspace function calls respectively.

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

5.3.1 Context-Aware Policy. If the policy for a given executable defines specific function call contexts for particular rules, `bpfbox` instruments these function calls using uprobes (for userspace functions) and kprobes (for kernelspace functions). Each instrumented function call is associated with a unique bit in the process' *state* bitmask. A probe is triggered on entry that causes `bpfbox` to flip the corresponding bit to a `1`, and again on return, flipping the corresponding bit back to a `0`. An inherent limitation of this approach is that it prohibits recursive function calls from being used to specify rule contexts, though we believe that this limitation should be acceptable in practice (see Section 9.1). In such cases, the policy can simply fall back to specifying ordinary rules.

5.4 Collecting and Logging Audit Data

When an operation is denied or matches with an audit rule, `bpfbox` submits an event to userspace for logging. To accomplish this, we leverage the new ringbuf map type added in Linux 5.8 [35], a ring buffer that is efficiently shared across all CPUs. In userspace, the `bpfbox` daemon uses `mmap(2)` to map the corresponding memory region and polls for new data at regular intervals. As events are consumed in userspace they are removed from the ring buffer to make room for new events. Since the ringbuf map provides strong order guarantees and high performance under contention, we can ensure that `bpfbox` always provides highly reliable and performant per-event auditing.

6 BPFBOX POLICY LANGUAGE

In this section, we discuss the bpfbox policy definition language. Our design goals for the bpfbox policy language were as follows:

Simplicity: To facilitate policy creation and auditability, bpfbox policies should be as short and simple as possible while remaining expressive enough to sandbox most applications effectively.
Application Transparency: bpfbox policies should be application transparent—defining and implementing a bpfbox policy should not require any modification to the source code of the target application.
Flexibility: bpfbox policies should be flexible enough such that it is possible to be define both fine-grained and coarse-grained policy according to the needs of the end user.
Security: bpfbox policies should be designed with security in mind, adhering to all basic security principles. It should be difficult to write an insecure policy.

In the rest of this section we describe the basic constructs and syntax used to define bpfbox policy and explain the rationale for our design decisions.

6.1 Writing bpfbox Policy

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

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

In the subsections that follow, we will describe the rule categories supported by bpfbox (Sections 6.1.1 to 6.1.4) and the directives that may optionally be used to decorate them (Sections 6.1.5 to 6.1.6). Listing 1 depicts a simple example bpfbox policy.

6.1.1 Filesystem Rules. Filesystem rules in bpfbox govern what operations a process may perform on filesystem objects such as files and directories. They are written as `fs("pathname", access)` where `"pathname"` is a string containing the pathname of the file and `access` is a list of one or more file access permissions joined by the vertical bar symbol (|). For instance, to represent read and append permissions on /var/log/my_log, the corresponding bpfbox rule would be `fs("/var/log/my_log", read|append)`. In total, bpfbox supports nine distinct filesystem access flags as shown in Table 1.

bpfbox supports a limited globbing syntax when defining pathnames, allowing multiple rules matching similar files to be combined into one. Although filesystem rules are specified using pathnames, bpfbox internally uses inode and device numbers rather

Listing 1: An example of a bpfbox policy.

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

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

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

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

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

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

Table 1: The filesystem access flags supported in bpfbox.

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

than the pathnames themselves. When loading policies, bpfbox automatically resolves the provided pathnames into their respective inode-device number pairs. This information is then used to look up the correct policy whenever a sandboxed application attempts to access an inode. Since bpfbox does not check the pathnames themselves when referring to files, it is able to defeat time-of-check-to-time-of-use (TOCTOU) attacks, where an attacker quickly swaps out one file with a link to another in an attempt to circumvent access control restrictions in a privileged (most often setuid) binary [4]. In such a situation, bpfbox would simply see a different inode and deny access.

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

6.1.2 Network Rules. bpfbox implements networking policy at the socket level, covering both Internet sockets as well as Unix domain sockets. Networking rules are specified using `net(protocol, access)`, where `protocol` is a networking protocol like inet, inet6, or unix and `access` is a list of socket operations (Table 2) separated by vertical bars. For example, a rule targeting bind, accept, and connect operations on an inet6 socket would look like `net(inet6, bind|connect|accept)`, while a rule targeting create operations on a unix socket would look like `net(unix, create)`.

Table 2: The socket operation flags supported in bpfbox.

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

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

6.1.4 ptrace Rules. Just like with signals, ptrace access is specified as `ptrace("exe", access)`, where `access` is a list of allowed ptrace modes separated by vertical bars. The `child` keyword is also available for ptrace rules to allow tracing of any child process, regardless of the child's current profile. For instance, a rule that allows a process to read and attach to a child process would be written as `ptrace(child, read|attach)`, while a rule that allows only read access to processes running /usr/bin/ls would be written as `ptrace("/usr/bin/ls", read)`. Note that currently ptrace rules do not override other ptrace restrictions, such as those imposed by Yama [8].

6.1.5 Allow, Taint, and Audit Directives. bpfbox supports three distinct directives for defining *actions* that should be taken when a given access matches a rule. The `#[allow]` directive causes bpfbox to allow the access; however, it is not typically necessary to explicitly specify this as undecorated rules are assumed to be allowed by default. Regardless, it may be desirable to decorate such rules with `#[allow]` to improve the clarity of the policy. `#[taint]` is used to mark a rule as a *taint rule*, which causes the process to enter a tainted state when matched. These rules can be thought of as gateways into the rest of the policy. Once a process is tainted, this cannot be reversed unless it makes an execve(2) call explicitly marked with `#[untaint]`. Finally, `#[audit]` may be combined with `#[allow]` to cause bpfbox to log the matching operation to its audit logs. This can be useful for marking rare behavior that should be investigated or for determining how often a given rule is matched in practice.

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

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

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

7 APACHE HTTPD: AN EXAMPLE BPFBOX POLICY

In this section, we present an example bpfbox policy for Apache's httpd webserver and offer a comparison to the AppArmor and seccomp-bpf profiles used by snap's httpd[4] policy. We hope that this policy should serve both as an illustrative example of bpfbox's capabilities as well as a motivating comparison between bpfbox policy and the state of the art in process confinement.

The bpfbox policy, depicted in Listing 2, begins by specifying the location of the Apache httpd executable as well as the conditions required for tainting the httpd process. These taint rules define the boundary between httpd's setup phase and its main work loop by instrumenting the point at which it binds to either an IPv4 or IPv6 network socket. If there is a taint rule, the rest of the policy file is only enforced once the taint condition is satisfied. In other words, with taint rules the policy only applies when the process begins processing untrusted input.

Listing 2: An example policy for the Apache httpd webserver.

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

/* Taint on binding to an inet or inet6 socket */
#[taint] {
    net(inet,  bind)
    net(inet6, bind)
}

/* Specify allowed network access */
net(inet, any)
net(inet6, any)
net(unix, create|connect|send|recv)
net(netlink, create|bind|send|recv)

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

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

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

/* Serve files from /srv/html/ and all subdirectories */
fs("/srv/html/**", read|getattr)

/* Read configuration */
fs("/usr/share/httpd/**", read|getattr)
fs("/etc/httpd/", getattr)
fs("/etc/httpd/conf/**", read|getattr)
fs("/usr/share/zoneinfo/**", read|getattr)

fs("/etc/resolv.conf", read|getattr)
fs("/etc/hosts", read|getattr)

fs("/proc/sys/kernel/random/boot_id", read)
fs("/proc/sys/kernel/ngroups_max", read)

fs("/usr/lib/httpd/modules/*.so", getattr|read|exec)
fs("/usr/lib/libnss*.so.*", getattr|read|exec)
fs("/usr/lib/libgcc_s.so.*", getattr|read|exec)

/* Transition to a separate suexec policy */
#[transition]
fs("/usr/bin/suexec", getattr|read|exec)
```

[4]The Nextcloud snap package: https://snapcraft.io/nextcloud

Once tainted, we specify a set of allowed socket operations, both for networking and for interprocess communication, as well the ability for httpd to send fatal signals to other processes running the same profile. The file system rules for httpd are slightly more granular. Some notable examples include the ability to append to log files, read configuration files, and read static content from /srv/http. In practice, this could be easily reconfigured according the needs of the end user, say to add support for a writable subdirectory.

This policy allows for the execution of all httpd shared library modules. If desired, this could be easily restricted to a subset of these modules by providing explicit pathnames. It also specifies that the suexec program, used by httpd to execute external programs with different privileges, should cause the process to transition to a separate profile. The suexec profile can then be used to control specifically which programs should be run and what transitions (if any) might occur as a result.

Note this policy works at the LSM level rather than the system call level, with all LSM hooks that are not explicitly allowed being denied. As a result, virtually all privileged operations are forbidden. Accesses to devices, kernel modules, other processes—these and more are excluded because they are not listed in the policy. By enabling direct access to process state and LSM, bpfbox policies work at an abstraction level that is both straightforward and precise.

7.1 A Comparison with Snap

Snap packages define their security policy [46] through an app declaration provided in the package's snapcraft.yml file (depicted in Listing 3). The package author lists the "apps" provided by their package and assigns attributes and policy plugins for each app. This coarse-grained policy is then translated into corresponding AppArmor and seccomp-bpf policy files according to the information specified in the app declaration. For comparison with the bpfbox policy in Listing 2, we will consider both the snapcraft.yml declarations as well as the generated policy files.

Listing 3: The relevant portions of httpd's snapcraft.yml app declaration.

```
apps:
  # Apache daemon
  apache:
    command: run-httpd -k start -DFOREGROUND
    stop-command: httpd-wrapper -k stop
    daemon: simple
    restart-condition: always
    plugs: [network, network-bind, removable-media]
```

In order to make writing policy for snap packages as easy as possible, policy definitions within the app declaration are very coarse grained, emphasizing simplicity and terseness over expressiveness. snap's httpd policy lists the network, network-bind, and removable-media policy plugins in addition to the default snap policy. While this approach undoubtedly makes it simple and easy to write policy, it does little to improve policy auditability; at a glance, it is unclear what effect each policy plugin has on the resulting security policy.

These snap declarations produce much more complex, often overly generalized AppArmor and seccomp-bpf policies. They are rather large, consisting of 494 lines of policy (ignoring blank and comment lines) for AppArmor and 410 lines for seccomp-bpf. Further, the generated seccomp-bpf policy contains several generic rules that allow `chown(2)` to change the owner and group of files to `root`, despite the fact that httpd runs under its own `httpd` user. Even worse, the generated AppArmor policy permits the execution of 120 common shell utilities, many of which are unnecessary for httpd's normal operation.

The result of attempting to combine a very high-level approach with two complex, low-level confinement mechanisms is three difficult-to-audit policy files. The app declaration is far too coarse-grained to be precisely sure what each policy plugin is doing, while the AppArmor and seccomp-bpf policy files are so fine-grained, verbose, and overly generalized that their auditability suffers as well. If this app declaration were replaced with a single `bpfbox` policy, we posit that it would not be significantly harder to write than the original app declaration while maintaining a steady advantage in auditability, as well as an advantage in terseness over the generated AppAmor and seccomp-bpf policy files.

8 PERFORMANCE EVALUATION

In this section, we present a series of performance benchmarks aimed at ascertaining the overhead imposed by `bpfbox` on both sandboxed applications and the system itself. Benchmarking data was collected using the Phoronix Test Suite [29], a comprehensive benchmarking platform for the Linux operating system, and an ad-hoc kernel compilation benchmark. To establish a metric for comparison against other process confinement mechanisms, we ran the same benchmarks under the AppArmor [12] mandatory access control LSM. All benchmarks were run under three distinct conditions, summarized below.

Base: No policy engine is running. This measures the baseline operation of the system.

Passive: The policy engine is running in the background without any active profiles. This measures the impact of the policy engine on unconfined processes.

Permissive The policy engine is running in permissive mode in the background with empty profiles for each benchmarking program. This represents the worst case scenario where the policy engine complains about (i.e., logs) *every* security-sensitive operation.

Tests were conducted on an x86_64 Arch Linux virtual machine running a Linux 5.8.0-rc6 kernel, with eight virtual CPUs running at 2.99GHz and 16GB of RAM. Our results show that `bpfbox` incurs acceptable overhead, on par with and in some cases better than AppArmor. The benchmarking results are presented in Table 3 and are summarized in the subsections that follow.

8.1 Phoronix OSBench Benchmarks

The Phoronix OSBench benchmarking suite provides a series of tests that measure the performance of various aspects of operating system functionality. In particular, it measures file creation, process creation, thread creation, program execution, and memory allocation. Data for each test was collected using `strict-run` mode, which prioritizes the accuracy of results, running as many tests as needed.

The results of the OSBench benchmark show that `bpfbox` imposes modest overhead on an unconfined system, with a difference of under 4% in all tests. The "Memory Allocations" tests actually show the `bpfbox` and AppArmor configurations slightly outperforming the base system; however, given they are all separated by less than one standard deviation (1.87% standard deviation for base), they should all be seen as essentially the same for this test. When `bpfbox` was actively logging all security-sensitive operations performed by the benchmarking programs, this overhead increased to about 15% at the highest. Predictably, the "Create Files" and "Launch Programs" tests exhibited the most overhead, as both test cases resulted in significantly more instrumentation work on the part of `bpfbox`.

Although AppArmor exhibited slightly less overhead on unconfined processes, the difference between the two is largely marginal, within a few microseconds. In the worst case, `bpfbox` performed significantly better than AppArmor in the "Create Files" and "Launch Programs" tests. This difference is likely attributable to the fact that `bpfbox`'s kernelspace logging mechanisms transfer data to userspace more efficiently.

8.2 Kernel Compilation Benchmarks

Kernel compilation benchmarks present an ideal test case for measuring overhead under a busy workload with `execve(2)` calls and significant disk I/O. This workload is more representative of the worst case for `bpfbox` overhead, especially when `bpfbox` is instrumenting the behavior of all involved processes. In particular, the "Permissive" phase of the benchmark involved the instrumentation of 15 distinct profiles and resulted in several thousand audit log events per second.

Each trial consisted of six timed Linux 5.8.0-rc6 kernel compilations, with the initial run discarded to eliminate transient disk I/O. Compilations were timed using the GNU `time(1)` command line utility.

In the kernel compilation benchmarks, `bpfbox` and AppArmor exhibit roughly equivalent overhead on unconfined processes. In the worst case, `bpfbox` exhibits significantly less overhead in kernelspace but loses out in total elapsed time by just over 130 seconds. This disparity is likely due to the fact that the current version of `bpfbox` does not buffer writes to its log file in userspace. The sheer volume of events generated by this benchmark causes `bpfbox`'s userspace logging component to consume significantly more CPU resources than in previous trials. This performance bottleneck can be fixed with a relatively straightforward change to `bpfbox`'s userspace code.

8.3 Phoronix Apache Benchmarks

The Phoronix Apache benchmark provides an excellent test case for measuring the performance impact of `bpfbox` on socket networking operations. This test runs the Apache httpd server and attempts as many requests as possible within a fixed amount of time. Higher requests per second results in a better score.

In this benchmark, `bpfbox` is shown to have a slightly higher base impact on the system than in previous benchmarks at just under 6%, but the worst-case "Permissive" impact on requests per second is quite modest at under 9%, a difference of just over 1000 requests

Table 3: The results of the Phoronix OSBench, kernel compilation, and Phoronix Apache benchmarks. Percent differences are given in parentheses. For the kernel compilation benchmarks, "User" and "System" represent total CPU time spent in userspace and kernelspace while "Elapsed" represents real elapsed time.

	Base	Passive				Permissive			
		bpfbox		AppArmor		bpfbox		AppArmor	
Phoronix OSBench (lower is better):									
Create Files (μs)	27.86	28.94	(3.81%)	28.01	(0.55%)	32.31	(14.80%)	96.56	(110.44%)
Create Threads (μs)	25.96	26.90	(3.56%)	26.28	(1.24%)	27.67	(6.39%)	26.09	(0.51%)
Launch Programs (μs)	75.12	78.02	(3.79%)	77.64	(3.30%)	87.31	(15.01%)	102.43	(30.76%)
Create Processes (μs)	51.32	52.53	(2.34%)	51.61	(0.57%)	51.85	(1.04%)	52.11	(1.54%)
Memory Allocations (ns)	113.98	112.33	(-1.45%)	112.29	(-1.50%)	112.75	(-1.09%)	112.74	(-1.09%)
Kernel Compilation (lower is better):									
User (s)	14457.01	14564.80	(0.74%)	14711.42	(1.74%)	14829.11	(2.54%)	14432.09	(-0.17%)
System (s)	1712.59	1760.02	(2.73%)	1765.69	(3.05%)	1804.10	(5.20%)	2544.72	(39.09%)
Elapsed (s)	2086.92	2114.83	(1.33%)	2130.38	(2.06%)	2397.48	(13.85%)	2261.09	(8.01%)
Phoronix Apache (higher is better):									
Requests Per Second (r/s)	14686.95	13887.59	(-5.59%)	13743.88	(-6.63%)	13504.23	(-8.39%)	13431.34	(-8.93%)

per second. bpfbox's performance here is roughly equivalent to AppArmor.

9 DISCUSSION

Our work began with the dual insight that eBPF was now capable of implementing kernelspace security mechanisms and that there was an opportunity to create a better solution for Linux process confinement. bpfbox serves as a proof of concept of the former and a significant argument for the latter. bpfbox is a research prototype, however, and as such it has a number of limitations and opportunities for enhancements. We believe it also shows that there are a number of unexplored opportunities in systems security that are made feasible with eBPF. We discuss each of these in the rest of this section.

9.1 Limitations

While our current implementation has a number of limitations, we see most of them as opportunities for future work rather than inherent limitations of our approach. For example, one limitation of our approach to sub-application level confinement (i.e., the `#[func]` directives discussed in Section 5.3.1 and Section 6.1.6) is that it breaks for recursive function calls. This limitation exists due to the nature of how uprobes work in eBPF. A uprobe temporarily replaces the address of a function call with a trap to the eBPF program which trampolines back to the original address when it terminates. Uretprobes work in much the same way, instrumenting the return address instead. This means that eBPF programs cannot distinguish between a return from a recursive function call and a return from a top-level function call. Since the current implementation of bpfbox relies on setting and unsetting per-process flags to manage state, this may lead to unexpected denials within such recursive function calls. Having said this, we do not believe this limitation to be significant in practice, and if it were a factor, several workarounds are possible including explicitly defined userspace tracepoints (USDTs).

Since bpfbox relies on instrumenting various eBPF programs in order to manage process-to-policy associations as well as enforce the policy itself, the current implementation of bpfbox is unable to sandbox processes that started before bpfbox has run, as no profile will be associated with those processes. To overcome this limitation, future iterations of bpfbox could scan the entries in /proc/{pid}/exe on start to manually perform the correct policy associations.

Despite the fact that bpfbox does not require modifications to the Linux kernel source code, it does require a compatible Linux version that has been compiled to support specific eBPF features. In particular, bpfbox depends on the KRSI LSM instrumentation framework (available in Linux 5.7+) and the ringbuf map type (available in Linux 5.8+), as well as a kernel that has been compiled with support for loading eBPF programs and exposes BTF (BPF Type Format) debugging symbols. Many mainstream Linux distributions come with this support enabled by default, while others require the user to manually set the appropriate configuration flags before compiling the kernel. It may be some time, however, before many systems upgrade to Linux 5.8+ kernels.

9.2 Potential Improvements

Perhaps the most obvious extension to bpfbox is the introduction of seccomp-like functionality, through which policies could specify access at the system call level in addition to the LSM hook level. eBPF already supports instrumentation of processes at the system call level through the syscalls:* and raw_syscalls:* tracepoint families. Unlike traditional seccomp-bpf, these filters could be applied *externally* to the target application and without relying on execution through a wrapper application (e.g., MBOX [27]) for application transparency. Filters could then be integrated with the existing LSM probe framework to provide one cohesive policy definition language rather than relying on two distinct solutions.

The extensibility and flexibility of eBPF is highly conducive to integration with various aspects of the operating system, including other security solutions. For instance, it may be possible to integrate future versions of bpfbox with intrusion detection and prevention systems, assigning specific detection thresholds to particular rules.

bpfbox could deny access to a specific set of files or prevent the execution of specific programs if the system detects an attack in progress. This integration could also be extended to other eBPF programs monitoring various other aspects of system behavior.

Rather than strictly focusing on policy enforcement at the application and sub-application levels, bpfbox could be extended to also enforce policy at the container level rather than just the process level. Policies could be written to harden containers and further restrict access to external system resources. eBPF has excellent support for the instrumentation of individual cgroups, so this would be a natural extension to the current bpfbox implementation.

Thanks to bpfbox's detailed event logging, it should be possible to integrate audit2allow-like [42] functionality such that profiles can be automatically generated through the analysis of permissive mode audit logs. This should allow even inexperienced stakeholders to quickly generate working bpfbox policies that capture normal application behavior. These policies should then be relatively easy to audit and extend, due to bpfbox's simple policy language syntax.

Although bpfbox currently depends on the Python 3 bcc toolchain and its LLVM backend, future versions may be ported to libbpf CO-RE [34] (Compile Once, Run Everywhere), a framework that uses BTF type information and an automatically generated header file to allow eBPF programs to be compiled once and distributed to any compatible Linux configuration. This will enable bpfbox to be deployed on embedded systems where the disk space and runtime overhead of its dependencies would otherwise be prohibitive.

9.3 Future Directions

eBPF provides a verifiably safe way to add functionality to the Linux kernel at runtime. Code can be triggered by and respond to a wide variety of kernel-level and userspace events. While this sort of code could always be written, eBPF simplifies its development: terminating the eBPF loading process will cause the eBPF code to be safely unloaded without requiring a system reboot. Further, eBPF's shared maps and buffered event output mechanisms allows for easy, performant data exchange between userspace and kernelspace. Many others are excited about the performance gains that eBPF can bring to applications [7, 11, 49]. We suspect, however, that eBPF may have an even bigger impact on security.

A central challenge in systems security is how to appropriately recognize and respond to security-critical events. Sometimes it is clear, as with opening a file, that an action has potential security implications. With complex systems, such as those underpinning large cloud infrastructures, security problems can arise in almost any context. As attacks evolve, defenders must evolve as well. Today this evolution manifests as a never-ending series of software updates. Applications must be updated because that is our primary means by which vulnerabilities can be mitigated. But what if we had other options?

Sandboxing is ultimately an aspiration rather than a technical mechanism. In principle, a proper reference monitor architecture would prevent untrusted code from ever violating system policy; in practice, shared code, shared state, and imprecise policies allow for numerous opportunities for sandbox escapes and privilege escalation. If we want to do better, we have to rethink our approach to the boundaries around our computations.

Exploits respect no fixed boundaries, yet standard security mechanisms do. The kernel does not directly control how processes run, and processes do not control the kernel; instead, they interact (primarily) through the system call interface. When security problems arise on one side of this divide, traditionally it must be solved on that side.

bpfbox demonstrates that, with technologies like eBPF, we don't have to be so rigid in our thinking. Applications can change how the kernel makes security decisions, and the kernel can directly manipulate how processess run to improve security. Implementing security mechanisms changes from an either/or problem—do we make security enhancements in userspace or kernelspace?—to both. We can cross the boundaries to mitigate vulnerabilities just as attackers cross boundaries to exploit them.

By exposing LSM hooks, bpfbox makes it feasible to add sandboxing to any process. It also potentially enables security mitigations that go beyond any conventional sandbox. Authorizations can be tied to functions in userspace and restrictions placed on core kernel functionality. Further, with modest extensions, security policy could be directly tied to program activity. In some contexts such as denial-of-service attacks, resource allocation can impact security. When allocations can be observed and manipulated system-wide and communicated to critical processes, resource allocations can be made in light of security priorities, rather than simply attempting to be fair to all requesters. The design space of potential mechanisms is very large, and we believe there is a huge opportunity in exploring it.

10 CONCLUSION

We have presented the design and implementation of bpfbox, a novel process confinement engine written entirely in eBPF using the KRSI framework. We leverage the flexibility of eBPF to provide finer-grained instrumentation on application behavior than previously possible and integrate this information with an extensible enforcement framework that enables simple yet expressive policies. We are confident that bpfbox can be extended to meet the confinement requirements of almost any application in practice.

State-of-the-art confinement solutions like snap and docker combine various complex and heterogeneous mechanisms for limiting access to system resources. What these approaches all lack is a unified solution built from the ground up for process confinement. Such a unified approach could potentially have significant benefits for cloud applications by allowing for simplified and more secure container management, even enabling new application deployment abstractions. We believe bpfbox is a step towards such a unified solution.

11 ACKNOWLEDGMENTS

This research is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) through a Discovery Grant.

REFERENCES

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. 1986. Mach: A new kernel foundation for UNIX development. In *Proceedings of Summer Usenix*. http://cseweb.ucsd.edu/classes/wi11/cse221/papers/accetta86.pdf

[2] bcc authors. 2020. *iovisor/bcc*. The IOVisor Project. https://github.com/iovisor/bcc

[3] Brian N. Bershad, Craig Chambers, Susan Eggers, Chris Maeda, Dylan McNamee, Przemyslaw Pardyak, Stefan Savage, and Emin Gün Sirer. 1994. SPIN: An Extensible Microkernel for Application-Specific Operating System Services. In *Proceedings of the 6th Workshop on ACM SIGOPS European Workshop: Matching Operating Systems to Application Needs* (Wadern, Germany) *(EW 6)*. Association for Computing Machinery, New York, NY, USA, 68–71. https://doi.org/10.1145/504390.504408

[4] Matt Bishop and Michael Dilger. 1996. Checking for Race Conditions in File Accesses. *Computing Systems* 9, 2 (1996), 131–152. https://static.usenix.org/publications/compsystems/1996/spr_bishop.pdf

[5] Bubblewrap authors. 2020. *Bubblewrap*. https://github.com/containers/bubblewrap

[6] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. 2004. Dynamic Instrumentation of Production Systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Boston, MA) *(ATEC '04)*. USENIX Association, Berkeley, CA, USA, 2–2. https://www.usenix.org/legacy/publications/library/proceedings/usenix04/tech/general/full_papers/cantrill/cantrill.pdf

[7] C. Cassagnes, L. Trestioreanu, C. Joly, and R. State. 2020. The rise of eBPF for non-intrusive performance monitoring. In *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*.

[8] Kees Cook. 2010. *[PATCH] security: Yama LSM*. https://lkml.org/lkml/2010/6/21/407

[9] Jonathan Corbet. 2007. Notes from a container. *LWN.net* (October 29 2007). https://lwn.net/Articles/256389/

[10] Jonathan Corbet. 2019. KRSI — the other BPF security module. *LWN.net* (December 27 2019). https://lwn.net/Articles/808048/

[11] Glauber Costa. 2020. *How io_uring and eBPF Will Revolutionize Programming in Linux*. https://thenewstack.io/how-io_uring-and-ebpf-will-revolutionize-programming-in-linux/

[12] Crispin Cowan, Steve Beattie, Greg Kroah-Hartman, Calton Pu, Perry Wagle, and Virgil Gligor. 2000. SubDomain: Parsimonious Server Security. In *Proceedings of the 14st Large Installation Systems Administration Conference (LISA)*. USENIX Association, New Orleans, LA, United States. https://www.usenix.org/legacy/event/lisa2000/full_papers/cowan/cowan.pdf

[13] Docker. 2020. Docker security. https://docs.docker.com/engine/security/security/

[14] Will Drewry. 2012. *Dynamic seccomp policies (using BPF filters)*. https://lwn.net/Articles/475019/

[15] Firejail authors. 2020. *Firejail*. https://firejail.wordpress.com/

[16] Flatpak authors. 2020. *Flatpak*. https://flatpak.org/

[17] Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. 2004. Ostia: A Delegating Architecture for Secure System Call Interposition.. In *Network and Distributed System Security (NDSS) Symposium*. Internet Society, San Diego, California. https://www.ndss-symposium.org/wp-content/uploads/2017/09/Ostia-A-Delegating-Architecture-for-Secure-System-Call-Interposition-Tal-Garfinke.pdf

[18] Ian Goldberg, David Wagner, Randi Thomas, and Eric Brewer. 1996. A Secure Environment for Untrusted Helper Applications (Confining the Wily Hacker). In *USENIX Security*. https://www.usenix.org/legacy/publications/library/proceedings/sec96/full_papers/goldberg/goldberg.pdf

[19] Brendan Gregg and Jim Mauro. 2011. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD* (1st ed.). Prentice Hall.

[20] Andreas Gruenbacher and Seth Arnold. 2007. AppArmor Technical Documentation. http://lkml.iu.edu/hypermail/linux/kernel/0706.1/0805/techdoc.pdf

[21] Toshiharu Harada, Takashi Horie, and Kazuo Tanaka. 2005. Towards a manageable Linux security. In *Linux Conference*, Vol. 2005. https://osdn.net/projects/tomoyo/docs/lc2005-en.pdf/en/2/lc2005-en.pdf

[22] Andrew Hurst. 2004. Analysis of Perl's taint mode. http://hurstdog.org/papers/hurst04taint.pdf

[23] Mahya Soleimani Jadidi, Mariusz Zaborski, Brian Kidney, and Jonathan Anderson. 2019. CapExec: Towards Transparently-Sandboxed Services. In *International Conference on Network and Service Management (CNSM)*. IEEE. https://doi.org/10.23919/CNSM46954.2019.9012736

[24] K Jain and R Sekar. 2000. User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement. In *NDSS*. https://www.cs.unc.edu/~fabian/course_papers/jain-userlevel.pdf

[25] Poul-Henning Kamp and Robert N M Watson. 2000. Jails: Confining the omnipotent root. In *2nd International SANE Conference*. http://ivanlef0u.fr/repo/madchat/sysadm/bsd/kamp.pdf

[26] Michael Kerrisk. 2013. Namespaces in operation, part 1: namespaces overview. *LWN.net* (January 4 2013). https://lwn.net/Articles/332974/

[27] Taesoo Kim and Nickolai Zeldovich. 2013. Practical and Effective Sandboxing for Non-root Users. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. USENIX Association, San Jose, CA, 139–144. https://www.usenix.org/system/files/conference/atc13/atc13-kim.pdf

[28] Butler W. Lampson. 1973. A note on the confinement problem. *Commun. ACM* 16, 10 (Oct. 1973), 613–615. https://doi.org/10.1145/362375.362389

[29] Michael Larabel and Matthew Tippett. 2011. *Phoronix Test Suite*. http://www.phoronix-test-suite.com/

[30] libbpf authors. 2020. *libbpf*. https://github.com/libbpf/libbpf

[31] J. Liedtke. 1995. On Micro-Kernel Construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain, Colorado, USA) *(SOSP '95)*. Association for Computing Machinery, New York, NY, USA, 237–250. https://doi.org/10.1145/224056.224075

[32] Steven McCanne and Van Jacobson. 1992. The BSD Packet Filter: A New Architecture for User-level Packet Capture. *USENIX Winter* 93 (1992). https://www.tcpdump.org/papers/bpf-usenix93.pdf

[33] James Morris, Stephen Smalley, Greg Kroah-Hartman, Chris Wright, and Crispin Cowan. 2002. Linux security modules: General security support for the linux kernel. In *USENIX Security Symposium*. ACM Berkeley, CA, 17–31. https://www.usenix.org/legacy/event/sec02/full_papers/wright/wright.pdf

[34] Andrii Nakryiko. 2020. *BPF Portability and CO-RE*. https://facebookmicrosites.github.io/bpf/blog/2020/02/19/bpf-portability-and-co-re.html

[35] Andrii Nakryiko. 2020. *BPF ring buffer*. https://lwn.net/Articles/820559/

[36] Pradeep Padala. 2002. Playing with ptrace, Part I. *Linux Journal* 2002, 103 (2002), 5. https://www.linuxjournal.com/article/6100

[37] David S Peterson, Matt Bishop, and Raju Pandey. 2002. A Flexible Containment Mechanism for Executing Untrusted Code. In *USENIX Security*. https://www.usenix.org/legacy/event/sec02/full_papers/peterson/peterson_html/

[38] Daniel Price and Andrew Tucker. 2004. Solaris Zones: Operating System Support for Consolidating Commercial Workloads.. In *LISA*, Vol. 4. 241–254. SolarisZones:OperatingSystemSupportforConsolidatingCommercialWorkloads

[39] Niels Provos. 2003. Improving Host Security with System Call Policies. In *USENIX Security*. https://www.usenix.org/legacy/events/sec03/tech/full_papers/provos/provos_html/

[40] Mickael Salaun. 2020. *landlock.io*. https://landlock.io/

[41] Seccomp authors. 2020. *Seccomp BPF (SECure COMPuting with filters)*. https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html

[42] SELinux authors. 2020. *SELinux Userspace Tools*. https://github.com/SELinuxProject/selinux

[43] Rui Shu, Peipei Wang, Sigmund A Gorski III, Benjamin Andow, Adwait Nadkarni, Luke Deshotels, Jason Gionta, William Enck, and Xiaohui Gu. 2016. A Study of Security Isolation Techniques. *Comput. Surveys* 49, 3 (Dec. 2016), 1–37. https://doi.org/10.1145/2988545

[44] KP Singh. 2019. *MAC and Audit policy using eBPF (KRSI)*. https://lwn.net/ml/linux-kernel/20191220154208.15895-1-kpsingh@chromium.org/

[45] Stephen Smalley, Chris Vance, and Wayne Salamon. 2001. Implementing SELinux as a Linux security module. *NAI Labs Report* 1, 43, 139. https://www.cs.unibo.it/~sacerdot/doc/so/slm/selinux-module.pdf

[46] Snapcraft. 2020. *Security policy and sandboxing*. https://snapcraft.io/docs/security-sandboxing

[47] Alexei Starovoitov and Daniel Borkmann. 2014. *Rework/optimize internal BPF interpreter's instruction set*. Kernel Patch. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=bd4cf0ed331a275e9bf5a49e6d0fd55dffc551b8

[48] Robert NM Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. 2010. Capsicum: Practical Capabilities for UNIX.. In *USENIX Security Symposium*, Vol. 46. 2. https://www.usenix.org/legacy/event/sec10/tech/full_papers/Watson.pdf

[49] zoidbergwill et al. 2020. *Awesome eBPF: A curated list of awesome projects related to eBPF*. https://github.com/zoidbergwill/awesome-ebpf