Programmable System Call Security with eBPF

Jinghao Jia\textsuperscript{1}, YiFei Zhu\textsuperscript{2}, Dan Williams\textsuperscript{3}, Andrea Arcangeli\textsuperscript{4}, Claudio Canella\textsuperscript{5}, Hubertus Franke\textsuperscript{6}, Tobin Feldman-Fitzthum\textsuperscript{6}, Dimitrios Skarlatos\textsuperscript{7}, Daniel Gruss\textsuperscript{8}, Tianyin Xu\textsuperscript{1}
\textsuperscript{1}University of Illinois at Urbana-Champaign, Urbana, IL, USA
\textsuperscript{2}Google, Inc., Sunnyvale, CA, USA
\textsuperscript{3}Virginia Tech, Blacksburg, VA, USA
\textsuperscript{4}Red Hat, Inc., New York, NY, USA
\textsuperscript{5}Amazon Web Services, Graz, Austria
\textsuperscript{6}IBM Research, Yorktown Heights, NY, USA
\textsuperscript{7}Carnegie Mellon University, Pittsburgh, PA, USA
\textsuperscript{8}Graz University of Technology, Graz, Austria

Abstract

System call filtering is a widely used security mechanism for protecting a shared OS kernel against untrusted user applications. However, existing system call filtering techniques either are too expensive due to the context switch overhead imposed by userspace agents, or lack sufficient programmability to express advanced policies. Seccomp, Linux’s system call filtering module, is widely used by modern container technologies, mobile apps, and system management services. Despite the adoption of the classic BPF language (cBPF), security policies in Seccomp are mostly limited to static allow lists, primarily because cBPF does not support stateful policies. Consequently, many essential security features cannot be expressed precisely and/or require kernel modifications.

In this paper, we present a programmable system call filtering mechanism, which enables more advanced security policies to be expressed by leveraging the extended BPF language (eBPF). More specifically, we create a new Seccomp eBPF program type, exposing, modifying or creating new eBPF helper functions to safely manage filter state, access kernel and user state, and utilize synchronization primitives. Importantly, our system integrates with existing kernel privilege and capability mechanisms, enabling unprivileged users to install advanced filters safely. Our evaluation shows that our eBPF-based filtering can enhance existing policies (e.g., reducing the attack surface of early execution phase by up to 55.4% for temporal specialization), mitigate real-world vulnerabilities, and accelerate filters.

1 Introduction

Modern computer systems run a large variety of untrusted applications on a trusted operating system (OS) kernel. These applications interact with the OS kernel through the system call interface. Hence, system call security is a cornerstone for protecting a shared kernel against untrusted user processes. System call filtering is a widely used system call security mechanism. The basic idea is to restrict the system calls a given process can invoke based on predefined security policies, thereby reducing the attack surface. The filtering is done at the entry point of every system call to decide whether to allow or deny each system call.

Early system call filtering techniques such as Janus [94] and Ostia [50] employ trusted userspace agents to implement system call security policies. However, userspace agents incur significant context switch overheads to system call filtering, because every system call needs to switch to user space for policy checking and then switch back. Furthermore, the implementations are prone to many common traps and pitfalls, such as time-of-check-to-time-of-use (TOCTTOU) based race conditions of argument values [22, 49, 84].

Modern system call filter techniques such as Linux’s Seccomp (SECure COMPuting) run entirely inside the OS kernel, without the overhead of additional context switches. This brings significant performance advantages over userspace agents and some resistance to TOCTTOU-based argument races. Today, Seccomp is widely used to provide “the most important security isolation boundary” [18]. For example, every Android app is isolated using Seccomp [83]; systemd uses Seccomp for user process sandboxing [29]; Google’s Sandboxed API project [14] uses Seccomp for sandboxing C/C++ libraries; Lightweight virtualization technologies such as Docker [15], Google gVisor [7], Amazon Firecraker [6, 18], LXC/LXD [9], Rkt [88], and Kubernetes [73] all use Seccomp.

However, a major limitation of Seccomp is the lack of sufficient programmability to express advanced security policies. From an initial mode (known as strict mode) that blocked all system calls except \texttt{read()}, \texttt{write()}, \texttt{.exit()}, and \texttt{sigreturn()}, Seccomp now (since Linux v3.5) allows custom security policies to be written in the classic BPF language (cBPF) [30] in the filter mode. Filter mode enables application-specific security policies and results in the wide adoptions of Seccomp by different applications.

Unfortunately, the programmability of cBPF is overly limited—security policies in Seccomp are mostly limited to static allow lists. This is primarily because cBPF provides no mechanism to store states and hence cBPF filters have to be stateless. Furthermore, cBPF provides no interface to invoke any other kernel utilities or other BPF programs. As a result, many desirable and/or essential system call filtering features cannot
be directly implemented based on Seccomp-cBPF, instead requiring significant kernel modifications (a major deployment obstacle). Section 3 discusses these features in details. Recognizing the need for more expressive policies, Seccomp recently added a new feature known as Notifier [23], to support the old idea of userspace agents, which unfortunately shares their limitations (performance overhead, TOCTTOU issues, etc.).

In this paper, we present a programmable system call filtering mechanism by leveraging the extended BPF language (eBPF). Our goal is to enable advanced system call security policies to better protect the shared OS kernel, without impairing the system call performance or reducing OS security.

Our choice of eBPF is a result of practicality considerations: 1) eBPF offers basic building blocks for the target programmability, including maps for statefulness and helper functions for interfacing with the kernel; 2) like cBPF, eBPF is verified to be safe by the kernel; and 3) we can largely reuse existing implementation code in Linux.

Note that naively opening the eBPF interface in Seccomp, as early Linux patches [36, 65, 102], is not a solution, because: 1) basic support is missing (e.g., synchronization primitives for serialization); 2) existing utilities (e.g., task storage) may not fit the Seccomp model because they target privileged contexts only; and 3) existing features are not safe for Seccomp use cases, e.g., the current user memory access feature cannot address TOCTTOU issues.

To this end, we create a new Seccomp-eBPF program type which is highly programmable for users to express advanced system call security policies in eBPF filter programs. Specifically, we expose, modify, and create new eBPF helper functions to safely manage filter state, access kernel and user state, as well as utilize synchronization primitives. Importantly, our system integrates with existing kernel privilege and capability mechanisms, enabling unprivileged users to install advanced filters safely and preventing privilege escalation. The security of Seccomp-eBPF is equivalent to the two existing kernel components: Seccomp and eBPF.

We implement the new Seccomp-eBPF program type on top of Seccomp in the Linux kernel. We maintain the existing Seccomp interface with tamper protection. We implement many features required by real-world use cases, such as checkpoint/restore in userspace (CRIU), sleepable filter, and deployment configuration to make Seccomp-eBPF a privileged feature. We also modified an existing container runtime (crun) to support Seccomp-eBPF-based system call filtering.

We use Seccomp-eBPF to implement various new security use cases, including system call count/rate limiting, flow-integrity protection (SFIP), and serialization. We show how these features can prevent real-world vulnerabilities that cannot be safely prevented by cBPF filters. We also use eBPF filters to enhance temporal specialization, which achieves up to 55.4% reduction of the system call interface of the early execution phases, compared to existing cBPF implementations.

Lastly, we use eBPF filters to implement validation caching which can improve application performance by up to 10%. The paper makes the following main contributions:

- We discuss several essential use cases of system call filtering, which reveal limitations of the state-of-the-art filtering mechanism exemplified by Seccomp.
- We present the design and implementation of Seccomp-eBPF that enhances programmability of system call security and integrates well with the kernel, without affecting performance or security.
- We implement and evaluate the advanced system call security features using Seccomp-eBPF filters for real-world applications against real-world vulnerabilities.

The code of our implementation of Seccomp-BPF on Linux can be found at,

https://github.com/xlab-uiuc/seccomp-ebpf-upstream

2 Background

In this section, we briefly discuss the necessary background for understanding the limitation of Seccomp as the state-of-the-practice system call filtering mechanism.

2.1 Seccomp-cBPF

Seccomp currently relies on the classic BPF (cBPF) language for users to express system call filters as programs [39, 71]. cBPF has a very simple register-based instruction set, making the filter programs easy to verify. Due to the limited programmability, cBPF filters in Seccomp mostly implement an allow list—the filter only allows a system call if the system call ID is specified. Occasionally, a cBPF filter will further check arguments of primitive data types and prevent a system call if the argument check fails. Pointer-typed arguments, however, cannot be dereferenced.

cBPF filters are stateless—the output of a Seccomp-cBPF filter execution depends only on the specified system call ID and argument values (in the allow list), because cBPF does not provide any utility of state management.

A cBPF filter is size-limited by 4096 instructions; therefore, complex security policies have to be implemented by a chain of multiple filters. All installed filters in the chain are executed for every system call, and the action with the highest precedence is returned. The chaining behavior, however, comes with a performance overhead mainly due to security mitigations (e.g., Spectre) against indirect jumps [74].

Under the cover, Seccomp chooses to transform cBPF filters into eBPF code internally to take the advantage of extensively optimized eBPF toolchains, which produce code that runs up to 4X faster on x86-64 over using cBPF directly [91]. Note: this does not mean system call filters can be implemented in eBPF—the language exposed is still cBPF and eBPF features (e.g., maps and helpers) are not available.
2.2 Seccomp Notifier

The limited expressiveness of cBPF makes it hard to implement complex security policies. Therefore, Seccomp recently incorporated support for a userspace agent (called Notifier [23]) to complement cBPF filters. Similar to early system call interposition frameworks [49, 50, 58, 94], it defers the decision to a trusted user agent. Specifically, when Seccomp captures a system call, it blocks the calling task and redirects the system call context (e.g., calling PID, system call ID, and argument values) to the agent.

A major disadvantage of Seccomp Notifier is its significant performance overhead due to the additional context switches introduced by switching to the user space back and forth. Furthermore, to examine the contents of system call arguments that are user-space pointers, Seccomp Notifier must use ptrace to access the memory of the monitored process. In addition to its performance implications, such inspection is subject to time-of-check-to-time-of-use (TOCTTOU) race conditions where a thread in the monitored program may change memory contents (and thus argument values) after the check has been completed by Seccomp Notifier. Finally, the need to run a userspace agent in a trusted domain makes it challenging to be used in some deployment environments, e.g., for daemonless container runtimes [12]. For all of these reasons, Seccomp Notifier is inadequate for complex system call filtering policies.

3 Essential System Call Filtering Features

We highlight four features, none of which can be implemented using Seccomp-cBPF, that will lead to a more efficient, and robust system call filtering framework: statefulness, expressiveness, synchronization, and safe user memory access. We motivate the need for each feature with concrete examples.

3.1 Statefulness

System call filtering with the existing Seccomp-cBPF framework is fundamentally stateless: an invocation of the system call filter cannot carry state to subsequent invocations. As a result, policies are overly loose. Here we describe three practical, tight policies that require state passing: system call count limiting, system call sequences checking, and enhanced temporal specialization.

System call count limiting. Many attacks rely on invoking specific system calls repeatedly to cause starvation (e.g., fork bomb) or overflows (§7.2.1). An effective defense is to limit the times a system call can be executed based on the demand of the application.

As a concrete example, a special case of system call count limiting is to restrict the use of a specific system call to a single invocation, which is a common goal for container runtimes. The goal is to prevent the launched containers from issuing exec() to replace the process image. Container runtimes take three common steps to launch a container: 1) installing Seccomp filters, 2) dropping privileges, and 3) launching containers using an exec system call. The three steps are exemplified by the following code from runc [13], a container runtime used by Docker and Kubernetes:

```c
1 // Without NoNewPrivileges, seccomp is a
2 // privileged operation, so we need to do
3 // this before dropping capabilities
4 if 1.cfg.Config.Seccomp != nil &
5   !1.cfg.NoNewPrivileges {
6   seccomp.InitSeccomp(...)  
7  }
8 // Drops the capabilities, sets the correct
9 // user and working dir, before executing the
10 // command inside the namespace
11 finalizeNamespace(...)
12 ...
13 pdeath.Restore()
14 ...
15 if unix.Getppid() != 1.parentPid ( ... )
16 ...
17 unix.Write(fd, []byte("0"))
18 ...
19 system.Exec(name, 1.cfg.Args[0:], os.Env())
```

The snippet shows that the Seccomp filter needs to be installed before calling exec (L19). The desired policy is to only allow exec once at L19, but not later. However, it is hard to implement the policy using stateless filters. Note that the filter installation (L6) cannot be moved to a later point, because it requires the CAP_SYS_ADMIN capability, which is dropped within finalizeNamespace (L11).

As a result, with cBPF filters, the dangerous exec system call is commonly allowed for the entire duration of the container execution, even though it is not needed by the containerized applications [4, 8].

In fact, exec is not an exception. The container runtime has to allow many other security-sensitive system calls in the same way, such as prctl, capset, and write as shown in the code snippet. As with exec, these system calls cannot be blocked in a stateless filter, either.

If the system call filtering framework supported stateful policies, the filter could keep track of the number of invocations of a (potentially dangerous) system call and block further invocations.

Checking system call sequences and state machines. Recent work [24] shows that an application’s system call behavior can be modeled by a “system call state machine”, which can be used for security enforcement named SFIP (Syscall Flow Integrity Protection); the state machine can be automatically generated using static analysis [24]. Moreover, prior work on IDS (Intrusion Detection Systems) has showed that an attack can be modeled by a sequence of system calls [27, 44, 45, 57, 77, 78]. Support for stateful filters that maintain sequences and state machines would enable a
Precise temporal specialization. Prior work [54, 63, 75] shows the benefits of fine-grained security policies for different execution phases of the target application which often need distinct sets of system calls.

Ghavamnia et al. [54] propose to apply cBPF Seccomp filters at different execution phases to achieve temporal system call specialization. However, this technique is fundamentally limited under the current Seccomp-cBPF security model. In Seccomp, a filter, once installed, cannot be uninstalled during the process lifetime. Filters installed in later phases are chained with the filters installed earlier; all the installed filters are executed for every system call and the most restrictive policy is applied. Hence, with cBPF Seccomp filters, a system call needed in Phase N has to be allowed in all earlier phases (Phases 1...N − 1), even though the system call is not needed in any of the N − 1 phases. Figure 4 illustrates this point with a two-phase temporal specialization (N = 2), where P1 (Initialization) has to include system calls from P2 (Serving) with Seccomp-cBPF.

The limitation is rooted in the fact that cBPF filters cannot record states (i.e., the current execution phase). By recording the current phase in a state variable and applying the corresponding policy, a stateful system call filtering framework would enable precise temporal specialization and achieve tighter security policies for each phase.

3.2 Expressiveness

As with any user-supplied code or policy running in the kernel, system call filtering frameworks typically must trade off expressiveness and safety. The safety goals of cBPF have led it to a design prioritizing safety, with an overly restricted instruction set and runtime.

Performance optimizations. Given the cBPF instruction set, a cBPF filter is typically in the form of an allow list, implemented by a series of conditional jumps [39, 71]. Complex policies result in long lists of jumps and multiple filters (due to the size limit of a cBPF filter, see §2). Consequently, system call checking becomes expensive due to the need of iterating over long jump lists [60, 65, 66, 71, 90] and the overhead of indirect jumps caused by mitigations to speculative vulnerabilities (e.g., Retpoline) [74]. To reduce the overhead, multiple optimizations were proposed, such as dedicated Seccomp caches [90], skip-list search [35], and filter merging [74].

A more expressive system call filtering framework can optimize the filter performance. First, advanced data structures with constant lookup time (e.g., a hash map) can be used to eliminate long jump lists. Such an implementation is essentially equivalent to the dedicated Seccomp cache [90].

Moreover, cBPF filters are limited by 4096 instructions; allowing more instructions could eliminate the overhead of indirect jumps caused by chaining multiple filters, raised by Retpoline or other mitigations to speculative vulnerabilities. In a stateful system call filtering environment, one can further use map entries to sequence filters for more complex policies with low overhead. As we will see in §5, such enhancements to expressiveness can be achieved without sacrificing safety.

Rate limiting. Besides the performance aspects, the limited expressiveness of cBPF, due to the simple instruction sets and the program size constraints, also makes it hard to support advanced policies. One such example is rate limiting which only allows specified system calls to be issued under expected rates. Rate limiting relies on a timer. However, there is no timer utility in cBPF; in fact, cBPF cannot invoke any kernel functions or utilities. Furthermore, cBPF cannot record the last time in any state variable. A system call filtering framework with better expressiveness could be a solution; in particular, by exposing advanced and complex operations to the filters in a safe way. Such a framework could expose the current time information to the filter to facilitate the desired system call rate limiting.

3.3 Synchronization

System call filtering frameworks essentially provide a platform for mandating access into the kernel and can be used to implement application- or system-wide policies to prevent misuse of the kernel. Specifically, there has been increasing reports on kernel vulnerabilities that manifest via race conditions and are exploitable by two concurrently executing system calls [59, 68, 97, 98, 100]. Mitigating such attacks requires kernel developers to identify the race condition in the kernel, patch the vulnerable code, potentially backport the revisions to older kernel versions, and release a patched version. The above process is time-consuming: waiting for the kernel patches could open a long window of vulnerability. A system call filtering framework which can serialize specific system calls that are known to be exploitable by system call racing can immediately and effectively nullify race conditions without waiting for patches, backports, and releases.

3.4 Safe User Memory Access

Since a large number of system calls take pointers to user memory as arguments, deep argument inspection (DPI) is long desired [40, 42]. Seccomp-cBPF cannot support DPI because it only checks non-pointer argument values, i.e., if an argument is a pointer, it cannot be dereferenced by Seccomp-cBPF, which means that accepting or rejecting the system call cannot depend on values in structures that are passed to system calls via pointers. In fact, it means that Seccomp-cBPF cannot even address any string values (e.g., a file path in open()). To enable DPI, a system call filtering framework needs to provide a safe way to access user memory referred to by the pointer arguments. The main challenge (which is
also the reason that Seccomp does not dereference pointers) is to avoid the time-of-check-to-time-of-use (TOCTTOU) issue [22, 49, 84], where user space can change the value of what is being pointed to between the time the filter checks it and the time the value gets used.

4 Threat Model and Design Goals

4.1 Threat Model

We strictly adhere to the current threat model of Seccomp. The goal is to restrict how untrusted userspace applications interact with the shared OS kernel through system calls to protect the kernel from userspace exploits (e.g., shellcode or ROP payload). The kernel is trusted.

Seccomp requires the calling context to either be privileged (having CAP_SYS_ADMIN in its user namespace), or set NO_NEW_PRIVS [38] which ensures that an unprivileged process cannot apply a malicious filter and then invoke a set-user-ID or other privileged program using exec.

Once a filter is installed onto a process, it cannot be removed before the process termination. A filter cannot be tampered—a filter program and its states will be invisible to unprivileged processes once it is installed.

4.2 Design Goals

Given this threat model, we set the following goals:

• **Expressiveness and kernel support.** We aim to support all the essential system call filtering features discussed in §3. This not only requires a more programmable and expressive language, but also additional kernel support.

• **Maintain Seccomp usage model and interfaces.** In order to provide a practical, familiar and useful system call filtering framework, we must adhere to the same usage and threat model of Seccomp (§4.1). Further, to lower the barrier of adoption, we aim to maintain its interface.

• **No privilege escalation for unprivileged users.** As Seccomp supports the unprivileged use case (§4.1), our design must ensure no privilege escalation.

5 Design

5.1 Overview

We develop a programmable system call filtering mechanism on top of Seccomp, by leveraging the extended BPF language (eBPF), to enable advanced security policies (see §3). eBPF is a fundamental redesign of the BPF infrastructure within the Linux kernel [5]. It not only has a rich instruction set and flexible control flows (e.g., bounded loops and BPF-to-BPF calls), but also offers new features, such as helper functions to interface with kernel utilities and maps as efficient storage primitives to maintain states.

The choice of eBPF is a result of practicality considerations: 1) eBPF offers basic building blocks for the target programmability, including maps for statefulness and helper functions for interfacing with the kernel; 2) eBPF is verified to be safe by the kernel; and 3) since Seccomp already converts cBPF code into eBPF internally (§2.1), we can largely reuse existing Seccomp implementation and workflow.

However, directly opening the eBPF interface in Seccomp is not a solution: 1) basic supports are missing (e.g., synchronization primitives for serialization); 2) existing utilities (e.g., task storage) may not fit the Seccomp model because they target privileged context only; and 3) existing features are not safe for Seccomp use cases, e.g., the current user memory access feature cannot address TOCTTOU issues.

We expose, modify, and create new eBPF helper functions to safely manage filter state, access kernel and user state, and utilize synchronization primitives. Figure 1 illustrates these helper functions. Importantly, our system integrates with existing kernel privilege and capability mechanisms, enabling unprivileged users to safely install advanced filters. Essentially, we create a new Seccomp-eBPF program type which is highly programmable to express advanced system call security policies in eBPF filter programs.

In terms of security, we reduce the security of Seccomp-eBPF to the security of Seccomp and eBPF.

5.2 Seccomp-eBPF Program Type

An eBPF program type defines what helper functions (helpers) a program of the type is allowed to invoke and the corresponding capabilities required for invoking them. In other words, a program type defines its interface to the kernel and the capability system to use the interface. For a given filter program, the eBPF verifier checks the helper invocation instructions in the filter program and the capabilities of the calling context at the load time, as shown in Figure 2.

Specifically, the eBPF verifier checks both eBPF language specifications and Seccomp-eBPF program type. We extend the eBPF verifier to check the new Seccomp-eBPF program type, BPF_PROG_TYPE_SECCOMP. Seccomp-eBPF restricts (1) the use of helper functions, (2) the access to Seccomp data structures. The eBPF verifier already provides hooks where we
directly add our verification code. For (1), we declare the helper functions that eBPF filters are permitted to use (§5.3). For (2), we verify that filters only access data within the boundary of Seccomp data structures with correct offset and size. We strictly follow the verification against eBPF language specifications.

Figure 2 depicts the workflow of loading, installing, and running a Seccomp-eBPF filter. The filter is first loaded into the kernel, where it is verified and optionally JIT-compiled, and then installed in Seccomp. After that, the installed eBPF filter will be invoked upon every subsequent system call.

5.3 Helper Functions

Figure 1 shows the helper functions of the Seccomp-eBPF program type (BPF_PROG_TYPE_SECCOMP). Overall, there are ten helpers in five categories: (1) state management based on helpers to access (lookup/update/delete) maps, (2) serialization which needs synchronization helpers, (3) user access for safely accessing user memory, (4) kernel access for invoking kernel utilities such as time, and (5) eBPF program features such as tail calls that allow more complex programs.

Five of them already exist in Linux and can be directly exposed to the Seccomp-eBPF program type. The others are not available. In this section, we discuss the new helpers that are created from scratch or by modifying existing ones.

5.3.1 Task Storage. Maps are the basic enabler to statefulness (§3.1). eBPF currently supports different types of maps, including array maps and hash maps.

One important map type is task storage, which provides primitives for policies that need to maintain separate states for different tasks executing the same eBPF filter (loaded by the same FD). eBPF already implements task storage map, which uses Linux tasks as keys to access the stored values. This guarantees that storage from different tasks does not collide. The task storage can be used through two helpers: (1) bpf_task_storage_get and (2) bpf_task_storage_delete.

However, both the two helpers require additional capabilities, CAP_BPF and CAP_PERFMON, and thus cannot be used in unprivileged contexts, which hinders Seccomp’s use cases (an unprivileged user process can install a Seccomp filter as long as it sets NO_NEW_PROCS, see §4.1). Our goal is to redesign the helpers for task storage maps to make them essential utilities of Seccomp-eBPF filters.

Why do the helpers for task storage maps need additional privileges, while the other map helpers do not (e.g., bpf_map_lookup_elem in Figure 1)? The reason is that the API design of the existing helpers for task storage maps makes them insecure to unprivileged Seccomp-eBPF filters: the existing helpers used to access the task storage require a task struct pointer as input argument to find the corresponding map storage. While a privileged eBPF filter can call bpf_get_current_task_btf to retrieve the needed task struct pointer, such operation is insecure in an unprivileged context because a task struct contains a pointer to its parent task struct. A malicious filter can dereference the parent pointer recursively to reach the init task (PID 0) from the current process, leaking sensitive kernel information.

To avoid this issue and provide unprivileged filters with secure task storage, we create a new set of task-storage helpers. Our new helpers do not require a task struct as input. Instead, the helpers automatically find the current task’s group leader. For a process, this is its own task struct; for a thread, this is the group leader—the task that spawned it.

5.3.2 Safe User Memory Access. Safely accessing user memory is an important feature for checking pointer-typed argument values, as discussed in §3.4. We support such a feature and expose it through specialized helpers. Note that eBPF provides two helpers for accessing memory of user processes. However, these helpers cannot be used for deep argument inspection because they cannot address TOCTTOU-based argument racing [49, 97].

We follow the prior work on deep argument inspection [25, 40]. The key principle is to disallow user space to modify the argument values during and after the values are checked. This can be achieved by (1) copying the target user memory into a protected memory region that is only accessible by the kernel, (2) making the target user memory write-protected or inaccessible to user space, or (3) using protection-key functionalities in the kernel to prevent races from user space [32, 47, 61]. We implement the first two solutions, as they can be done on all commodity hardware, and expose them with user-memory access helpers.

Capability. The existing user-memory access helpers from eBPF require CAP_BPF and CAP_PERFMON. We reduce the security of Seccomp-eBPF to Seccomp Notifier [23] which allows the userspace agent to read and copy user memory if the agent is allowed to ptrace the process. i.e., the security of Seccomp Notifier is equal to ptrace. Therefore, we also reduce the capabilities of user-memory access helpers to ptrace.
Linux determines if one process (tracer) can ptrace another process (tracee) based on the following checks: 1) they are in the same thread group, 2) they are under the same user and group, or if the tracer has CAP_SYS_PTRACE, 3) the tracee cannot be traced without CAP_SYS_PTRACE if it has set itself to be non-dumpable, and 4) other LSM hooks.

How does ptrace apply to Seccomp? In Seccomp, the filter is regarded as the tracer, with the process that installs the filter being the tracee. Since unprivileged Seccomp requires the NO_NEW_PRIVS attribute to be set on the calling task, the UID/GID and capability set can not change after the filter installation. Hence, the in-kernel ptrace checks are largely covered by the NO_NEW_PRIVS attribute. Furthermore, the dumpable attribute is respected.

On Linux, the capability of ptrace also depends on a system-wide configuration, ptrace_scope [16]. To ensure that user-memory access helpers follow ptrace security, we define a new LSM hook that enforces the helpers to adhere to the policy set by ptrace_scope.

Protecting non-dumpable process. On Linux, a process (e.g., OpenSSH) handling sensitive information can set the "dumpable" attribute (via PR_SET_DUMPABLE) to prevent being coredumped or ptraced by another unprivileged process. Hence, an unprivileged Seccomp-eBPF filter should not be able to access memory of non-dumpable processes. To protect such processes, we apply a privilege requirement similar to ptrace. The helpers are modified to allow a filter to access non-dumpable memory only if the loader process has ptrace privileges (CAP_SYS_PTRACE).

Handling page faults. The original user-memory access helpers in Linux do not handle page faults when reading memory from userspace. The helpers use non-blocking functions (e.g., copy_from_user_nofault), which emit errors upon page faults. This is based on the dated assumption that a BPF program never sleeps, i.e., it cannot be blocked in the middle of execution [33]. This is inconvenient for Seccomp-eBPF filters—invoking a user-memory helper would fail if the memory access triggers a page fault. We support sleepable Seccomp-eBPF filters (§6). Therefore, our user-memory access helpers can handle page faults.

5.3.3 Serializing System Calls. The basic idea to serialize two racing system calls is to record the event of involved system calls. When a system call is invoked concurrently with another system call under execution which is known to have race vulnerabilities with it, the kernel stops the former system call until the latter finishes.

In this use case, the kernel records whether a particular system call (in terms of system call ID) is currently being executed and stores the information. For each system call, we add an integer atomic variable in the kernel to store whether there exists processes that are currently executing the system call. This atomic variable has an initial value of 0 and will be incremented by our new helper function. When a system call exits, its atomic variable is decremented, but the value will have a minimum of 0.

Helper API. We expose a new helper function, void bpf_wait_syscall(int curr_nr, int target_nr) for system call serialization. The helper function holds the execution of the current task until the execution of the target system call finishes. To achieve this, it increments the atomic variable of the current system call and perform busy waiting via a schedule loop until the atomic variable of the other system call is decremented to 0, which means no processes are executing the other system call. The helper is unprivileged, because it only affects the processes that attach the filter.

Enforcement. The eBPF filter that implements serialization uses a map to store system calls that have race vulnerabilities as key-value pairs. The map can be updated by a trusted userspace process after filter installation, when new race vulnerabilities need to be patched (§3.3). The filter implements a logic that, for each incoming system call, it checks whether the system call has a potential race condition based on the map. If so, the filter will invoke the bpf_wait_syscall helper to serialize the current system call.

To enforce system-wide serialization policies, we install the filter onto the init process. In Seccomp, a process inherits the filter of its parent process. Since init is the ancestor of all subsequent processes, the installed filter is inherited by all processes on the system, hence implementing a global policy. Since the kernel requires the root privilege when retrieving map file descriptor, only trusted, privileged processes can update the map.

5.4 Usage

The user loads a Seccomp-eBPF filter into the kernel via the bpf() system call and installs it using the seccomp() system call. Different from cBPF filters which are implemented using BPF instructions, the kernel expects eBPF filters to be loaded as bytecode. This allows eBPF filters to be written in high-level languages, such as C and Rust, and compiled using LLVM/Clang. In fact, eBPF has more mature and actively developing tool chains than cBPF (eBPF "is frozen [93]").

We add a new flag, SECCOMP_FILTER_FLAG_EXTENDED, to the seccomp() system call; if it is set, the filter is interpreted in eBPF; otherwise, it is in cBPF. Note that, different from cBPF filters, before installing an eBPF filter, one needs to load it using bpf(). Figure 3 shows the code snippets of loading and installing a Seccomp-eBPF filter, compared with installing a Seccomp-cBPF filter.

Note that cBPF does not have a separate load step (despite its name, the bpf() system call is specific to eBPF). A cBPF filter is installed in one step through the seccomp() system call, where the verification is done at the installation time. We choose to minimize changes to the existing seccomp() and bpf() interfaces, mainly for reducing adoption obstacles.
As a Seccomp-eBPF filter is verified and installed in two steps (§5.4), we develop the following two tamper protections.

### 5.5 Tamper Protection

As a Seccomp-eBPF filter is verified and installed in two steps (§5.4), we develop the following two tamper protections.

**Protecting filter program and maps.** For eBPF filters, `seccomp()` automatically closes the file descriptor (FD) of the eBPF program before returning to the user space. The user space is responsible for closing all FDs of maps before issuing the `seccomp` system call. This is to prevent leaks of the FDs, as anyone with access to the maps can potentially manipulate filter’s behavior. Our new function in libbpf, `bpf_seccomp_close_fd`, closes all these FDs except the eBPF program FD (which is needed as a parameter for `seccomp()` and gets closed inside).

However, the separation requires additional efforts to protect filters and maps against tampering (§5.5).

### Namespace tracking.

Currently, none of the helper functions we expose to Seccomp-eBPF filters requires additional privileges. If there is a need to expose helpers that require `CAP_BPF` and/or `CAP_PERFMON` (like many existing helpers), we need to enforce that an eBPF filter is loaded and installed in the same user namespace. Otherwise, an unprivileged attacker in the current user namespace can create a new user namespace (which has all capabilities by default [11]) to bypass the capability checks in the verifier, and then sends the filter FD back to its restricted parent namespace. We develop the enforcement by recording a reference to the load-time user namespace with the filter. When installing a filter, the kernel checks whether the current user namespace matches the recorded load-time user namespace.

### 6 Implementation

For practical uses in a variety of deployment environments (such as container environments), we addressed a number of implementation challenges.

**Sleepable Seccomp filters.** Sleepable filters are needed for Seccomp-eBPF (eBPF programs do not need to sleep), e.g., for page fault handling when accessing user memory (§5.3.2). To support sleepable filters, we create new BPF section names (seccomp and seccomp-sleepable). The BPF section is an ELF section that is used by libbpf to determine the sleepable attribute of the BPF program and set the flag when calling `bpf()`. This allows us to maintain the same `bpf()` interface without additional flags or other tooling support.

**Checkpoint/restore in userspace (CRIU).** CRIU is widely used by container engines to checkpoint the state of a running container to disk and restore it later. It facilitates features such as live migrations or snapshots. Seccomp currently supports CRIU only for eBPF filters. To support eBPF filters, we add two new utilities for (1) returning a file descriptor (FD) associated with an eBPF filter and (2) checkpointing map states by returning the FD of the n-th map.

Note that CRIU is only possible for privileged applications, such as container engines, as it requires `CAP_SYS_ADMIN`. Hence, an attacker taking control of an unprivileged application cannot retrieve the FD associated with a filter and modify them. Additionally, this mirrors the behavior of Seccomp-cBPF, where CRIU is considered secure.

**Container runtime integration.** Seccomp is an essential building block for modern container frameworks [48, 60]. It has been mentioned that Seccomp-eBPF filters are desired [70]. Integrating Seccomp-eBPF with existing container frameworks is straightforward, as our implementation maintains the same Seccomp interface.

To demonstrate this, we have integrated Seccomp-eBPF filter support in crun [89], a fast OCI-compliant container runtime and the default container runtime of Podman [12]. The code can be found at [89]. Attaching an eBPF filter to a Podman container can be done with the following command:

```
1  podman --runtime /usr/local/bin/crun run
2    --annotation run.oci.seccomp_ebpf_file
3      =ebpf_filter.o
```

Contrary to cBPF filters, which are generated from JSON-based profiles in existing container runtimes, eBPF filters are directly loaded into the kernel during the container initialization. Adding such support to other container runtimes, such as runc and CRI-O, can be done in a similar way.

**Seccomp-eBPF filters as a privileged feature.** For deployments that do not enable unprivileged eBPF, we provide a `sysctl` configuration to make Seccomp-eBPF a privileged feature.
feature. With it set, Seccomp-eBPF can only be used by processes with the \texttt{CAP\_SYS\_ADMIN} capability. Even with the configuration set, Seccomp-eBPF is still very useful for container runtimes and other management contexts (e.g., \texttt{init}) that run under the root privilege. For example, most of the container frameworks are not rootless.

7 Evaluation
We evaluate the usefulness and performance of Seccomp-eBPF. For usefulness, we use Seccomp-eBPF filters to enhance temporal system call specialization over existing cBPF filter implementations (§7.1). Moreover, we use eBPF filters to implement advanced security features that cannot be supported by cBPF filters. We evaluate them with real-world vulnerabilities listed in Table 2. We then evaluate the performance of eBPF filters with both micro and macro benchmarks. We also use eBPF filters to accelerate stateless checks (§7.3).

All experiments were run on an Intel i7-9700k with 8 cores, 3.60 GHz, and 32GB RAM. The machine runs Ubuntu 20.04 with Linux kernel v5.15.0-rc3, patched with our implementation of Seccomp-eBPF.

7.1 Precise Temporal Specialization
We explained how eBPF filters could enhance the security of temporal system call specialization (§3). This section quantifies the security benefits of implementing temporal specialization in eBPF filters and comparing them with existing cBPF filter implementations [54]. We evaluate temporal specialization for two distinct phases—P1 (initialization) and P2 (Serving), as illustrated in Figure 4.

cBPF filter. With cBPF filters, we install two filters: the first filter, installed at process startup, allows \( S_{\text{init}} + S_{\text{serv}} \); the second filter only allows \( S_{\text{serv}} \) and is installed at the program location that marks the start of the serving phase (the location is provided by the application developer). Hence, cBPF filters cannot precisely implement temporal specialization, i.e., only allowing \( S_{\text{init}} \) for the initialization phase. As discussed in §3, this problem becomes worse with more phases—a system call that is needed in the last phase has to be allowed in all the earlier phases.

eBPF filters. We implement temporal specialization in a single eBPF filter, installed at application startup, in which a global variable is used to record the phase. The eBPF filter strictly applies different policies based on the phase—it allows only \( S_{\text{init}} \) for the initialization phase and only \( S_{\text{serv}} \) for the serving phase. Hence, it addresses the limitations of cBPF filters. The following snippet sketches this filter:

```c
1     unsigned int phase;
2     SEC("seccomp")
3     int bpf_prog_ts(struct seccomp_data *ctx) {
4     if (!phase && ctx->nr == -1) {
5         phase = 1; // phase change
6         return SECCOMP_RET_ALLOW;
7     } else if (!phase) { // only allow S_init
8         switch (ctx->nr) {
9             case SYS_alarm:
10                 return SECCOMP_RET_ALLOW;
11             default: break;
12         }
13     } else { // only allow S_serv
14         switch (ctx->nr) {
15             case SYS_accept:
16                 return SECCOMP_RET_ALLOW;
17             default: break;
18         }
19     }
20     return SECCOMP_RET_ERRNO | EPERM;
21 }
```

Figure 4. Two-phase temporal specialization implemented with cBPF and eBPF filters. \( S_{\text{init}} \) and \( S_{\text{serv}} \) refer to the set of system calls required by the initialization and serving phase, respectively; \( S_{\text{comm}} = S_{\text{init}} \cap S_{\text{serv}} \)

Table 1. The numbers of unique system calls in different phases of the evaluated applications

| Application | \( |S_{\text{init}}| \) | \( |S_{\text{serv}}| \) | \( |S_{\text{comm}}| \) | Total | Init. Reduction |
|-------------|----------------|----------------|----------------|-------|---------------|
| HTTPD       | 71             | 83             | 47             | 107   | 33.6%         |
| NGINX       | 52             | 93             | 36             | 109   | 52.3%         |
| Lighttpd    | 46             | 78             | 25             | 99    | 53.5%         |
| Memcached   | 45             | 83             | 27             | 101   | 55.4%         |
| Redis       | 42             | 84             | 33             | 93    | 54.8%         |
| Bind        | 75             | 113            | 55             | 135   | 44.4%         |

Table 1 shows the numbers of unique system calls in different phases of the evaluated applications.

7.2 Advanced Security Features
Different from cBPF filters, where the second filter needs to be installed at the phase-changing location, we insert a dummy system call that marks the phase change.

Attack surface reduction. We use the six server applications in the temporal specialization work [54]. We use \( S_{\text{init}} \) and \( S_{\text{serv}} \) of each application, provided by the research artifact [54], from which we derive \( S_{\text{comm}} \). Table 1 shows the attack surface reduction of the initialization phase achieved by eBPF filters over the cBPF filters. The eBPF filters reduce
We present new security features enabled by Seccomp-eBPF filters to implement the kernel enforcement of system call filters and show that they can effectively mitigate real-world vulnerabilities (Table 2).

### 7.2 New Security Features

We present new security features enabled by Seccomp-eBPF filters and show that they can effectively mitigate real-world vulnerabilities (Table 2).

#### 7.2.1 Count and rate limiting

We implement eBPF filters for count limiting which only allows a system call to be invoked a limited number of times (§3). The filter keeps the count of the target system call using a global variable and increments the count when the system call is invoked. Once the count reaches a predefined value, all subsequent invocations of the specific system call are denied.

The following code shows an eBPF filter that only allows an application to call the keyctl system call with the argument KEYCTL_JOIN_SESSION_KEYRING up to a max allowed count—most applications (e.g., e4crypt) only need to call keyctl once or twice.

```c
unsigned int count;
SEC("seccomp")
int bpf_prog_cnt(struct seccomp_data *ctx) {
    if (ctx->nr == SYS_keyctl &&
        ctx->args[0] ==
        KEYCTL_JOIN_SESSION_KEYRING) {
        if (count >= MAX_ALLOWED_CNT) {
            return SECCOMP_RET_ERRNO | EPERM;
        }
        count += 1;
    }
    return SECCOMP_RET_ALLOW;
}
```

The eBPF filter can mitigate vulnerabilities like CVE-2016-0728 which is exploited by repeatedly calling keyctl with KEYCTL_JOIN_SESSION_KEYRING to create invalid keyring objects. This would trigger buggy error handling code that omits to decrement the refcount and thus cause the refcount to overflow. We also evaluate count limiting with CVE-2019-11487 and CVE-2017-5123 shown in Table 2.

Also, we implement rate limiting for frequency of specific system calls, using timer helpers (Figure 1).

#### 7.2.2 Flow integrity protection

We use Seccomp-eBPF filters to implement the kernel enforcement of system call flow-integrity protection (SFIP) [24] which otherwise requires kernel revisions and a new system call. SFIP checks both system call sequences and origins (code address that issues a system call). For the sequence check, we store the system call state machine in a two-level eBPF map, by encoding the state machine as a $N \times N$ matrix where $N$ is the number of system calls an application uses. We maintain the previous system call ID $s'$ in a global variable. Given a system call $s$, we check whether $s' \rightarrow s$ is valid. For the origin check, we store the system call origin mapping in another two-level map where the first-level map indexes the system call ID and the second-level stores the valid code addresses. For a system call $s$, we check whether $s$'s origin is included in the second-level map after indexing the first-level map with $s$'s ID (Seccomp already provides the calling address of a system call in its data structure).

We evaluate Seccomp-eBPF filter-based SFIP using the same buffer overflow vulnerability in BusyBox [2] in the evaluation of the original SFIP work [24]. The exploits require a system call sequence of either socket $\rightarrow$ execve to execute shellcode, or socket $\rightarrow$ mprotect to mark the memory protection of the shellcode as executable. Neither of the transitions is allowed by the eBPF filter based on the in-map system call state machines. Furthermore, the code addresses are checked as per the origin map.

#### 7.2.3 Serialization

We implement the eBPF based serializer (see §5.3.3) to mitigate the race-based vulnerabilities in Table 2. We write applications that issue the system calls of the race vulnerabilities concurrently. Take CVE-2018-18281 as an example, the filter serializes mremap and ftruncate when the applications issue both.

#### 7.3 Performance

We compare the performance of eBPF filters with eBPF filters and Notifier that implement the same security policy. For fair comparison, we use policies that can be implemented by eBPF filters. We expect the performance of eBPF filters and eBPF filters to be similar, since Seccomp internally converts eBPF filters into eBPF code. On the other hand, eBPF filters and eBPF filters go through different toolchains and thus are optimized differently.

<table>
<thead>
<tr>
<th>Vulnerabilities</th>
<th>Pattern</th>
<th>Involved System Call(s)</th>
<th>eBPF Filter as Defenses</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2016-0728</td>
<td>Repeated system calls</td>
<td>keyctl</td>
<td>Counter limiting</td>
</tr>
<tr>
<td>CVE-2019-11487</td>
<td>Repeated system calls</td>
<td>io_submit</td>
<td>Counter limiting</td>
</tr>
<tr>
<td>CVE-2017-5123</td>
<td>Repeated system calls</td>
<td>waitid</td>
<td>Counter limiting</td>
</tr>
<tr>
<td>BusyBox Bug #9071</td>
<td>System call sequences</td>
<td>(socket $\rightarrow$ exec) or (socket $\rightarrow$ mprotect)</td>
<td>Flow-integrity protection</td>
</tr>
<tr>
<td>CVE-2018-18281</td>
<td>Raced system calls</td>
<td>mremap, ftruncate</td>
<td>Serialization</td>
</tr>
<tr>
<td>CVE-2016-5195</td>
<td>Raced system calls</td>
<td>(write, madvice) or (ptrace, madvice)</td>
<td>Serialization</td>
</tr>
<tr>
<td>CVE-2017-7533</td>
<td>Raced system calls</td>
<td>fsnotify, rename</td>
<td>Serialization</td>
</tr>
</tbody>
</table>

Table 2. Evaluated vulnerabilities and their patterns that can be effectively defended by Seccomp-eBPF filters.
We use temporal specialization as the security policy. The latter generates a filter that sorts the system call ID and the corresponding argument values of a system call into a binary tree. We implement the eBPF filter in C and libseccomp optimize the filter code using binary search to avoid walking over a long jump list. Constant-action bitmap [31] is an optimization that skips the filter execution if the system call ID is known to be allowed. For Seccomp Notifier, we implement a userspace agent in C with the same logic as the eBPF filter. Our results show that the eBPF filter outperforms the unoptimized cBPF filter. It has roughly the same performance as the cBPF filter optimized by libseccomp. Both Clang and libseccomp optimize the filter code using binary search to avoid walking over a long jump list. Constant-action bitmap achieves the highest performance, but it cannot help policies on argument values. With Seccomp Notifier, getppid runs 45.4 times slower than with the eBPF filter.

7.3.2 Application Performance. We measure the application performance with different types of Seccomp filters. We use temporal specialization as the security policy. The implementations of the cBPF filters and eBPF filters are explained in §7.1. Seccomp Notifier can also support precise temporal specialization like eBPF filters. We implement a version that defers all decisions on system calls to a user agent. The handler keeps a phase-changing flag, which is set when the application enters the serving phase. We also evaluate an optimization that combines cBPF filter and the agent, denoted as hybrid. The hybrid version installs a cBPF filter at the process startup, allowing $S_{init}$. For every system call outside of $S_{init}$, the filter defers the decision to the user space. An additional filter is installed at the beginning of the serving phase to block $S_{init} - S_{comm}$. The handler then only allows $S_{serv} = S_{comm}$ after the phase change.

Applications and benchmarks. We use the same six server applications as in §7.1. We use official benchmark tools for each application (Table 4). All experiments are run 10 times and the average numbers are reported. Figure 5 shows the average latency and throughput of each application with cBPF, eBPF, Notifier, and Hybrid. We normalized all results to the baseline, a version that does not use Seccomp. The results show two consistent characteristics. First, Seccomp-eBPF has similar performance impacts as Seccomp-cBPF, while providing higher security for the initialization phase. Second, userspace agents incur significant performance overhead. Applications with pure user notifiers have additional 48%–188% average latency and 32%–65% lower throughput. Even for Hybrid, there is an additional 5%–108% average latency and 5%–52% lower throughput. NGINX suffers from the highest degradation, with latency increasing by a factor of 1.88 and throughput decreasing by 65%. Only Hybrid for Redis has a performance comparable to BPF filters, because in the serving phase of Redis, most system calls are from $S_{comm}$, which are filtered by cBPF filters.

7.3.3 Accelerating stateless security checks. Lastly, we use eBPF filters to implement the Draco Seccomp cache [90] and repeat the Draco evaluation using the applications and benchmarks in Table 4. The key idea of Draco is to cache the ID and the corresponding argument values of a system call that has recently been validated by stateless security checks. If an incoming system call hits the cache, Draco saves the computation of running the stateless checks. Draco is effective when the stateless checks are expensive and the system

### Table 3. Execution time of system calls with different types of filters and filter execution time.

<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
<th>Benchmark</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTPD</td>
<td>Web server</td>
<td>ab [1] (40 clients)</td>
</tr>
<tr>
<td>NGINX</td>
<td>Web server</td>
<td>ab [1] (40 clients)</td>
</tr>
<tr>
<td>Lighttpd</td>
<td>Web server</td>
<td>ab [1] (40 clients)</td>
</tr>
<tr>
<td>Memcached</td>
<td>In-memory cache</td>
<td>memtier [10] (10 threads)</td>
</tr>
<tr>
<td>Redis</td>
<td>Key-value store</td>
<td>memtier [10] (10 threads)</td>
</tr>
<tr>
<td>Bind</td>
<td>DNS server</td>
<td>dnseval [3]</td>
</tr>
</tbody>
</table>

### Figure 5. Avg. latency and throughput of the applications with different filters that implement temporal specialization (normalized to the baseline that disables Seccomp).

### Table 4. Applications and benchmarks used in evaluation.

<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
<th>Benchmark</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTPD</td>
<td>Web server</td>
<td>ab [1] (40 clients)</td>
</tr>
<tr>
<td>NGINX</td>
<td>Web server</td>
<td>ab [1] (40 clients)</td>
</tr>
<tr>
<td>Lighttpd</td>
<td>Web server</td>
<td>ab [1] (40 clients)</td>
</tr>
<tr>
<td>Memcached</td>
<td>In-memory cache</td>
<td>memtier [10] (10 threads)</td>
</tr>
<tr>
<td>Redis</td>
<td>Key-value store</td>
<td>memtier [10] (10 threads)</td>
</tr>
<tr>
<td>Bind</td>
<td>DNS server</td>
<td>dnseval [3]</td>
</tr>
</tbody>
</table>

### Table 3. Execution time of system calls with different types of filters and filter execution time.

<table>
<thead>
<tr>
<th>getppid (cycles)</th>
<th>filter (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No filter</td>
<td>244.18</td>
</tr>
<tr>
<td>cBPF filter (default)</td>
<td>493.06</td>
</tr>
<tr>
<td>cBPF filter (optimized)</td>
<td>329.47</td>
</tr>
<tr>
<td>eBPF filter</td>
<td>331.73</td>
</tr>
<tr>
<td>Constant-action bitmap [31]</td>
<td>297.60</td>
</tr>
<tr>
<td>Seccomp notifier</td>
<td>15045.05</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HTTPD</th>
<th>0.50</th>
<th>0.75</th>
<th>1.00</th>
<th>0.00</th>
</tr>
</thead>
<tbody>
<tr>
<td>NGINX</td>
<td>1.50</td>
<td>2.50</td>
<td>3.00</td>
<td>0.50</td>
</tr>
<tr>
<td>Lighttpd</td>
<td>0.50</td>
<td>0.75</td>
<td>1.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Memcached</td>
<td>0.75</td>
<td>1.00</td>
<td>1.50</td>
<td>0.00</td>
</tr>
<tr>
<td>Redis</td>
<td>1.50</td>
<td>2.00</td>
<td>2.50</td>
<td>0.50</td>
</tr>
<tr>
<td>Bind</td>
<td>0.00</td>
<td>0.50</td>
<td>1.00</td>
<td>1.50</td>
</tr>
</tbody>
</table>
calls of an application have high locality. The eBPF filter implements Draco using an array map which maps a system call to its corresponding check filter through an eBPF tail-call. The check filter uses a hash map to store recently validated argument values in a 48-byte blob.

We use the profile which only allows a system call if its ID and argument values are recorded by \texttt{strace} in a dry run. Our results show that eBPF-based Draco increases the throughput of three web servers by 10% on average. We do not observe significant improvement for the other applications.

8 Discussion

System call filtering with LSM. Linux Security Modules (LSMs) provide enforcement for system-wide security policies. Therefore, it is commonly discussed together with Seccomp, often as an alternative to implement system call filtering, especially with LSM-eBPF [41]. In fact, Seccomp-eBPF and LSM-eBPF are fundamentally different.

Seccomp restricts the system call entry point, while LSMs provide access control on kernel objects deeply on the path of system call handling. Hence, LSMs cannot prevent vulnerabilities before reaching LSM hooks. Moreover, Seccomp can provide fine-grained per-process system call filtering, while LSMs are system-wide. Lastly, LSM requires privileged use cases (only sysadmins can apply LSM policies); Seccomp-eBPF supports unprivileged use cases where an unprivileged process can install a Seccomp-eBPF filter.

Seccomp and LSM have different principles. Seccomp requires no new kernel code; hence, it stays future-proof for new syscalls; LSMs would need new code when new system calls are added. So, code in a Seccomp-eBPF filter takes a “microkernel” style. LSM uses a “monolithic kernel” style to implement custom checks for different system calls.

Automatic Seccomp-eBPF filter generation. Many tools have been developed to automatically generate application-specific Seccomp-eBPF filters. The basic idea is to profile the system calls of the target application with static code analysis [26, 53, 54, 82], binary analysis [26, 35], or dynamic tracking [87, 95]. The identified system calls are included in a static allow list.

How to automatically generate Seccomp-eBPF filters is an open question. Recent work like SFIP [24] shows promises of generating system call state machines. We believe that eBPF filters for count/rate limiting and temporal specialization can also be automatically generated. Moreover, many ideas from system call based intrusion detection using data modeling and machine learning can potentially be applied to generate advanced eBPF filters [27, 44, 45, 57, 77–79].

Risks of unprivileged eBPF. With bugs in Linux eBPF verifier and JIT compiler [81], Seccomp-eBPF may potentially allow unprivileged attackers to craft malicious eBPF filters to exploit vulnerabilities. Seccomp-cBPF shares the same risk; however, since it is simpler it likely has fewer bugs. We believe that in the long term unprivileged eBPF will be safe, as evidenced by recent work on formally-verified eBPF verifiers and compilers [51, 52, 81, 96, 99], and more broadly, safe and correct kernel code [21, 43, 76, 80].

Note that Seccomp-eBPF can be configured as a privileged feature at deployment (§6), if unprivileged eBPF is a concern. The privileged configuration can be used by many container runtimes and management services (e.g., \texttt{init}).

9 Related Work

System call filtering. Prior work has studied system call filtering (aka interposition) techniques for protecting the shared OS kernel against untrusted applications [17, 19, 34, 37, 46, 50, 56, 58, 67, 72, 77, 85, 86, 94]. Early techniques rely on userspace agents (e.g., based on \texttt{ptrace}) that check user-specified policies to decide which system calls to allow or deny, in the same vein as Seccomp Notifier (see §2.2). However, the context switch overhead could be unaffordable to applications that require high performance. Seccomp provides a solution that allows user-specified policies to be implemented in eBPF and executed as kernel extensions. Compared with userspace agents, Seccomp filters have significant performance advantages, which is one main reason that makes it a widely-used building block for modern sandbox and container technologies. Our work builds on the success of Seccomp and rethinks the programmability of system call security in the Seccomp context.

Discussions on eBPF Seccomp filters in the Linux community. There were a few other proposals on supporting eBPF filters for Seccomp, with patches [36, 65]. However, our discussions with the community both on the kernel mailing list [102] and at the Linux Plumbers Conference [69] tell that it is still very controversial.

One common concern is lacking concrete use cases, as exemplified by the maintainer’s response—“\textquote{What’s the reason for adding eBPF support? Seccomp shouldn’t need it... I’d rather stick with eBPF until we have an overwhelmingly good reason to use eBPF}...” [28]. One reason is that early patches [36, 65] do not support maps and thus still have no statefulness. We hope that our work addresses this concern. Our design is driven by an analysis of desired system call filtering features, which reveals the limitations of Seccomp (§3). We also show that existing eBPF utilities are insufficient; therefore, simply opening the eBPF interface cannot solve the problem.

There are other concerns on eBPF security, including (1) exposure of kernel data and functions to untrusted user space via maps and helpers, and (2) potential vulnerabilities in the eBPF subsystem. In our design, the security of Seccomp-eBPF can be systematically reduced to that of Seccomp and eBPF. We discuss the risk of unprivileged eBPF in §8. We believe that vulnerabilities in the eBPF subsystem implementation is a temporal (but challenging) problem that will be addressed...
in the longer term. Our implementation also provides a configuration option to turn Seccomp-eBPF into a root-only feature (§6) which is useful for container environments.

**Other eBPF use cases.** Recently, eBPF has been actively used to build innovative tools and systems, ranging from networking [64] to tracing [62] to storage [53, 101] to virtualization [20]. Different from most of the eBPF use cases, Seccomp needs support unprivileged use cases (§4.1). Therefore, security is a first-class design principle of Seccomp-eBPF.

## 10 Concluding Remarks

Our work makes one step towards empowering advanced system call security policies by enhancing the programmability of system call security mechanisms, namely Seccomp in Linux. We present our design and implementation of the Seccomp-eBPF program type and show that it can enable many useful features, without impairing system call performance or reducing system security. Our work is imperfect, but we hope that it could lay the foundation and motivate strong programmability for system call security.

### Acknowledgement

This work was supported in part by the IBM-Illinois Discovery Accelerator Institute and by NSF grant CNS-1956007. We thank Giuseppe Scruvano for helping with the eBPF Seccomp filter support in crun. We thank Kele Huang, Yicheng Lu, Austin Kuo, and Josep Torrellas for early participation of the work. We thank Michael Le, Mimi Zohar, and Hani Jamjoom for the discussions of the work. We also thank developers from the Linux community for discussing the work with us on the Linux kernel mailing list and at LPC 2023.

### References

[28] Kees Cook. Re: [PATCH net-next 0/3] `eBPF Seccomp filters. https://lore.kernel.org/netdev/CAGXu5iLY90RSRUj_2xLB03Wo5G07njp0ESR4Smnsm1UsEw@mail.gmail.com/, February 2018.
[34] Asit Dan, Ajay Mohindra, Rajiv Ramaswami, and Dinkar Sitara. ChakraVyuha (CV): A Sandbox Operating System Environment for


Tom Hromatka. Using a cBPF Binary Tree in Libseccomp to Improve Performance. In Linux Plumbers Conference (LPC’18), November 2018.


