# **POSTER: Transparent Temporally-Specialized System Call Filters**

Matthew Rossi matthew.rossi@unibg.it Università degli Studi di Bergamo Bergamo, Italy

Dario Facchinetti dario.facchinetti@unibg.it Università degli Studi di Bergamo Bergamo, Italy

#### Abstract

Reducing the attack surface of the OS kernel is an effective technique to enhance the security of application workloads. In Linux systems, developers can restrict the set of available system calls by using seccomp. Although being widely adopted in browsers, container runtimes, and sandboxing tools, this approach presents some challenges: (i) applying precise filters often requires significant application modifications, which can impede developer productivity, and (ii) the transparent enforcement of filters is bound to use a single, static list with every syscall the application might ever need, resulting in overly permissive and less effective security boundaries.

In this paper, we propose an automated method to generate temporally-specialized seccomp filters tailored to the current application state. This significantly enhances the effectiveness of filters, and overcomes the major limitations associated with a single, static filter. We implement our solution by leveraging the eBPF subsystem in the Linux kernel. Specifically, we use in-kernel eBPF programs to monitor the application state and dynamically enable or disable specialized seccomp filters in response to state transitions. We discuss how this approach addresses the limitations of state-of-the-art solutions. Finally, we validate the feasibility of our proposal and show that it introduces a limited overhead.

#### **CCS** Concepts

 Security and privacy → Software and application security; Access control.

#### Keywords

Syscall filtering, Temporal specialization, Attack surface reduction, eBPF, Seccomp

#### **ACM Reference Format:**

Matthew Rossi, Michele Beretta, Dario Facchinetti, and Stefano Paraboschi. 2025. POSTER: Transparent Temporally-Specialized System Call Filters. In ACM Asia Conference on Computer and Communications Security (ASIA CCS '25), August 25–29, 2025, Hanoi, Vietnam. ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3708821.3735342

ASIA CCS '25, Hanoi, Vietnam

© 2025 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-1410-8/2025/08 https://doi.org/10.1145/3708821.3735342 Michele Beretta michele.beretta@unibg.it Università degli Studi di Bergamo Bergamo, Italy

Stefano Paraboschi stefano.paraboschi@unibg.it Università degli Studi di Bergamo Bergamo, Italy

# 1 Introduction

Several research works (e.g., [7, 11]) have shown that access to unnecessary syscalls increases the risk of privilege escalation and correlates with a higher frequency of zero-day vulnerabilities. The reason is that less commonly used kernel APIs are more susceptible to bugs, whereas popular ones are more robust and tested [12].

The introduction of *seccomp* [1] represents a key advancement in safeguarding the kernel from potentially vulnerable unprivileged user space applications. Indeed, seccomp enables the specification of a syscall filter that is evaluated by the kernel with minimal overhead whenever an application invokes a syscall. This mechanism is widely employed by various applications, such as browsers, container runtimes, and sandboxing tools. However, it also introduces some challenges. Indeed, once activated, seccomp filters can only be further restricted, requiring the implementation of tight security boundaries through significant application restructuring. To become seccomp-aware, an application must separate its functions into distinct compartments and manage different security profiles at runtime. In practice, most (if not all) seccomp users apply the filter once, often at application startup, and never change it afterwards. This means that all required syscalls are included in a single, overly permissive filter, which exposes a wider attack surface, and diminishes the security benefits. Another issue is that developers often struggle to craft effective syscall filters, since they typically operate at higher level of abstractions (i.e., developers directly invoke library APIs rather than syscalls).

To improve the current scenario, novel research [9, 10] has introduced the promising concept of *temporally-specialized* filters that are tailored around specific application compartments. Despite being innovative, both works have shortcomings. For instance, Ghavamnia et al. [9] provide tools to generate the filters, but still require the developer to restructure the application manually. Jia et al. [10] instead greatly enhance seccomp's flexibility, but their work requires several kernel architectural changes.

This paper advances the state-of-the-art with an approach to enforce temporally-specialized filters that does not require application changes nor kernel modifications. The design relies on the eBPF kernel subsystem, and permits to transparently apply filters to the application based on its current state. In addition, we provide the developer with tools to automatically generate the filters associated with every application state. To this end, the developer is only required to identify functions that trigger state changes (e.g., any

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

functions that transitions a web server from an initialization to a serving phase), and to run the application in a test environment.

In the following we detail our approach and discuss the experimental evaluation, which confirms the merits of our solution and its limited performance overhead.

# 2 Background

This section provides a concise overview of eBPF, detailing the essential information needed to understand the rest of this paper.

eBPF [5] is a Linux subsystem that allows programs to run within the kernel in response to the execution of kernel functions. Specifically, eBPF programs are attached to designated hook points, and are executed whenever these are triggered. This allows for the inspection of the hook's input arguments and, for functions allowing error injection, the modification of the return value. Essentially, these programs permit to alter the kernel behavior without changing its source code or introducing custom modules. For the purposes of this paper it is important to note that (i) data structures called maps can be used to maintain state across multiple invocations of eBPF programs and to share data with user space, and (ii) eBPF programs loaded into the kernel undergo rigorous validation to attest their safety (e.g., a program cannot crash due to memory errors). Modern eBPF development is supported by *frontends* such as libbpf, i.e., frameworks that allow developers to write eBPF programs in a C-like dialect which is then converted into cross-platform bytecode.

### 3 Our approach

This section clarifies the threat model and presents our solution. We begin with a high-level overview, followed by detailed explanations of the generation and enforcement of filters.

*Threat model.* Similarly to the seccomp framework, our proposal limits the set of system calls available to user space applications. We consider the kernel trusted, although it may be affected by vulnerabilities. Our goal is to reduce the attack surface and better defend the kernel against an attacker gaining remote code execution due to vulnerabilities in unprivileged user space applications.

#### 3.1 Overview

From a high-level perspective, we replace static filters, traditionally set by manually invoking the seccomp syscall, with dynamic filters enforced through eBPF. This approach requires loading a set of eBPF programs that are evaluated whenever syscalls are invoked. Note that loading eBPF programs into the kernel is a privileged operation, as it demands the CAP\_BPF capability, and hence we delegate this task to a system administrator. This operation occurs only once at application deployment, and simply involves executing the application with a provided loader.

The uploaded eBPF programs operate either (i) in *tracing* mode, in which they collect all syscalls executed by the application and categorize them by application state, enabling the generation of filters, or (ii) in *enforcement* mode, in which the previously generated filters are activated in response to state transitions. Filters are recorded (during tracing) and loaded (during enforcement) in dedicated maps. The architecture of our solution is shown in Figure 1.



Figure 1: Tracing and sandboxing of a target application

#### 3.2 Monitoring the application lifetime

The implementation of our approach presents two primary technical challenges: (i) tracing the processes of the target application, and (ii) detecting transitions in the application's state.

Processes are traced by injecting in the kernel a set of eBPF programs to monitor the application's lifetime. These programs are attached to the sched\_process\_fork and the sched\_process\_exit tracepoints, which are triggered by both the fork and clone syscalls, and also at process termination. Using the application's main thread id as a starting point, the eBPF programs track all child processes by adding or removing their ids in a TYPE\_HASH map.

To detect transitions we rely on eBPF's ability to probe user space processes. Specifically, a second set of eBPF programs are activated when the application executes a transitioning function. These programs maintain the mapping between application threads and states. Every thread is associated with a single state, but multiple threads with their respective states and syscalls may coexist at runtime. We consider the definition of state transition functions a developer-provided input, which entails sharing its name with the system administrator. No application changes are required.

## 3.3 Generating the filters

To generate the filters, the developer runs the application in a test environment with a provided binary, which (i) prepares a process for the application's execution, (ii) records its process id in the application monitoring maps detailed in § 3.2, and (iii) launches the application via an execve syscall. An eBPF program is attached to the tp\_btf/sys\_enter tracepoint to capture all the syscalls issued by the application. When executed, it first reconstructs the mapping between thread id and application state using the monitoring maps, and then extracts the requested syscall id, storing it in a backing array. Upon application termination, this information is retrieved by the tracer to generate the corresponding filters (see Figure 1).

#### 3.4 Activating the filters

The generated filters (§ 3.3) are applied at runtime without modifying the application's structure or manually invoking the seccomp syscall. The developer just provides the system administrator with the application binary and the generated filters. The system administrator then executes the application binary using the provided

 Table 1: Performance of local web servers without syscall filtering, with seccomp, and with our eBPF-based solution

Software	P99 latency [ms]			Throughput [req/s]		
	Native	Seccomp	eBPF	Native	Seccomp	eBPF
Apache 2.4.58	18.64	19.11	19.51	3.46K	2.72K	2.39K
Lighttpd 1.4.74	2.79	2.91	3.34	5.45K	5.23K	4.69K
Nginx 1.24.0	6.32	6.58	6.96	3.09K	2.99K	2.93K

loader (§ 3.1). This process is sufficient to enforce the filters automatically at runtime, and requires no additional actions.

Filter enforcement is handled by a separate eBPF program attached to kernel probes that monitors all syscalls issued by the application's threads. If a thread's application state does not allow a specific syscall, i.e., it is absent from the backing array representing the filter, the eBPF program invokes the bpf\_override\_return helper function to inject an error. This means that the kernel code implementing the syscall is not run at all, and instead a permission denied error is returned, resulting in a failed syscall invocation.

#### 4 Evaluation

To test our approach, we first isolated the overhead introduced by eBPF with a microbenchmark that measures the performance of the lightweight getpid syscall, and then evaluated the impact of syscall filtering on popular web servers (Apache, Lighttpd, Nginx). In both experiments, we compared a test without protections (native case) against the use of seccomp and our eBPF-based solution. Experiments were conducted on an Ubuntu 24.04 server, kernel 6.8.0, an AMD 7985WX CPU, 256 GiB of DDR5 RAM, and 2 TB SSD.

*Microbenchmark.* In this preliminary experiment we measured the time to execute the getpid syscall over 1 million invocations. The average time was 207 *ns* when running without protections, 241 *ns* (+16.4%) with seccomp, and 309 (+28.2% w.r.t. seccomp) with our solution. This result was expected, since getpid is one of the shortest-lived syscalls, and because seccomp is currently the most efficient approach to filter syscalls directly within the kernel.

Web server. In this test we evaluated the overhead introduced by syscall filtering techniques on web servers. In detail, we used Wrk<sup>1</sup> to request the default web server page for 30 seconds, using 100 connections parallelized over 12 threads. Table 1 reports the 99th percentile of latency and the average throughput in the three different configurations. The result we obtained are promising: comparing our solution with seccomp, latency increased by 2.1% (-12.1% throughput) for Apache, by 14.7% (-10.3% throughput) for Lighttpd, and by 5.8% (-2.0% throughput) for Nginx.

#### 5 Related Work

While seccomp remains a fundamental technique to safeguard the kernel, recent research has explored precise *temporal specialization* [9, 10] to enhance its effectiveness, with the goal of enabling fine-grained seccomp filters, tailored to specific application states. Although innovative, these approaches have limitations. For instance, Ghavamnia et al. [9] propose the generation of temporallyspecialized policies, but the solution requires developers to make applications seccomp-aware. Jia et al. [10] significantly enhance seccomps's programmability, but introduce several kernel architectural changes, hindering adoption. Extensive research [7, 13, 14] has also investigated user space solutions. This offers great flexibility, but it incurs substantial overhead due to frequent context switches between kernel and user space. A promising binary rewriting technique named *zpoline* [15] can be used to avoid them, but unfortunately it cannot filter syscalls issued by dynamic libraries.

The automatic generation of syscall filters has also been widely studied. Proposed methods fall into two main categories: static and dynamic generators. Static generators [6, 8] extract the syscalls required by an application directly from its code, whereas dynamic generators [2–4] are based on runtime application monitoring. Both approaches have limitations. For instance, static generators struggle with interpreters or managed runtimes, while the coverage of dynamic generators depends on the runtime tests conducted.

#### 6 Conclusions and future work

This paper proposed an approach to transparently apply temporallyspecialized seccomp filters, without requiring modifications to the application or to the kernel. Preliminary results confirmed the introduction of a limited overhead. In future work we aim to explore various hooking strategies and investigate the integration of our solution into orchestration frameworks.

#### Acknowledgments

This work was supported in part by the EC under project GLACIA-TION (01070141), by the Italian MUR under PRIN project POLAR (2022LA8XBH), and by projects SERICS (PE00000014) and GRINS (PE00000018) in the NRRP MUR program funded by the EU–NGEU.

#### References

- [1] 2025. Seccomp BPF. https://docs.kernel.org/userspace-api/seccomp\_filter.html
- [2] 2025. SlimToolkit. https://slimtoolkit.org/
   [3] M. Abbadini, M. Beretta, D. Facchinetti, G. Oldani, M. Rossi, and S. Paraboschi.
- 2023. Lightweight Cloud Application Sandboxing. In *CLOUDCOM*.
   [4] M. Abbadini, D. Facchinetti, G. Oldani, M. Rossi, and S. Paraboschi. 2023. NatiSand:
- [4] Marboadin, D. rachinetti, G. Ordan, W. Rossi, and S. Faladostin. 2022. Rational. Native Code Sandboxing for JavaScript Runtimes. In *RAID*.
   [5] J. Corbet. 2014. *BPF: the universal in-kernel virtual machine*. https://lwn.net/
- [5] J. Corbet. 2014. BY: the universal in-kernet virtual machine. https://lwi.net/ Articles/599755/
- [6] N. DeMarinis, K. Williams-King, D. Jin, R. Fonseca, and V.P. Kemerlis. 2020. Sysfilter: Automated system call filtering for commodity software. In *RAID*.
- [7] T. Garfinkel, B. Pfaff, and M. Rosenblum. 2004. Ostia: A delegating architecture for secure system call interposition. In NDSS.
- [8] S. Ghavamnia, T. Palit, A. Benameur, and M. Polychronakis. 2020. Confine: Automated system call policy generation for container attack surface reduction. In *RAID*.
- [9] S. Ghavamnia, T. Palit, S. Mishra, and M. Polychronakis. 2020. Temporal system call specialization for attack surface reduction. In USENIX.
- [10] J. Jia, Y. ZhuFei, D. Williams, A. Arcangeli, C. Canella, H. Franke, T. Feldman-Fitzthum, D. Skarlatos, D. Gruss, and T. Xu. 2023. Programmable system call security with eBPF. arXiv (2023).
- [11] V. Kemerlis, Vasileios P., M. Polychronakis, and A. D. Keromytis. 2014. ret2dir: Rethinking kernel isolation. In USENIX.
- [12] Y. Li, B. Dolan-Gavitt, S. Weber, and J. Justin. 2017. Lock-in-Pop: Securing privileged operating system kernels by keeping on the beaten path. In USENIX ATC.
- [13] C. Linn, M. Rajagopalan, S. Baker, C. Collberg, and S. Debraya nd J.H. Hartman. 2005. Protecting Against Unexpected System Calls. In USENIX.
- [14] S. Pailoor, X. Wang, H. Shacham, and I. Dillig. 2020. Automated policy synthesis for system call sandboxing. OOPSLA (2020).
- [15] K. Yasukata, H. Tazaki, P.L. Aublin, and K. Ishiguro. 2023. zpoline: a system call hook mechanism based on binary rewriting. In USENIX ATC.

<sup>&</sup>lt;sup>1</sup>https://github.com/wg/wrk