MAKING SAFE OS KERNEL EXTENSIONS EXPRESSIVE, EFFICIENT, AND EASY TO PROGRAM

BY

JINGHAO JIA

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2025

Urbana, Illinois

ABSTRACT

OS Kernel extensibility has long been an essential capability of contemporary operating systems. In recent years, *safe* kernel extensions have gained significant traction, evolving from simple packet filters to large, complex programs that customize storage, networking, and scheduling. As the name suggests, these frameworks promise important safety properties for extension programs, including the absence of crashing behavior and proper termination. A notable example is the eBPF kernel extensions on Linux, which employs a static verification scheme to ensure safety properties hold on all program paths. With the increasing popularity, problems and issues in existing safe kernel extension frameworks gradually emerge. This dissertation dives into these problems for three important aspects of safe kernel extensibility—expressiveness, efficiency, as well as programmability—and identifies them as consequences of improper tradeoffs in the design of safe kernel extensions.

On the expressiveness aspect, this dissertation looks at the use case of system call filtering with Linux's Seccomp, where the existing cBPF extension fails to provide the needed expressiveness to support advanced system call security use cases. This dissertation proposes a new, programmable system call filtering mechanism, Seccomp-eBPF, which leverages the more expressive eBPF extensions through a carefully designed framework and security model. Using Seccomp-eBPF, users, whether privileged or not, are able to implement enhanced system call security policies to mitigate vulnerabilities that cannot be handled before.

Next, on the efficiency aspect, this dissertation considers two instrumentation use cases of kernel extensions: kernel probing and kernel control flow integrity protection, both suffering from the inefficiency of existing mechanisms. For the kernel probing use case, this dissertation presents Uno-kprobe, a new kernel probe design that achieves universally trapless probing and eliminates the expensive context switches associated with existing trap-based probing implementation for kernel extensions. For kernel control flow integrity protection, this dissertation introduces eKCFI, an eBPF-based KCFI framework that features new instrumentation and attachment mechanisms, which achieves complete kernel code coverage, scalable eBPF program attachment, and efficient kernel instrumentation.

Lastly, on the programmability aspect, this dissertation focuses on the existing eBPF extension framework, where the use of static verification incurs significant difficulties in programming eBPF programs—correct programs are often rejected by the verifier due to a gap between the high-level programming language and the low-level verification. This dissertation proposes a new design of safe kernel extension frameworks with Rex, in which extension safety is realized through the inherent language-based safety from Rust, combined with a lightweight extralingual runtime for properties unsuitable for static analysis. Rex eliminates the need for a static verifier and the associated efforts to work around verification issues, thereby closing the language-verifier gap and significantly improving extension programmability.

To those who have been there seeing me and supporting me—we dreamed alone, but together we believed.

ACKNOWLEDGMENTS

As I sit down to write my dissertation, I still cannot believe that I have already been a Ph.D. student for five years and am about to leave. In these five years, I have met and interacted with many wonderful people and made lots of good memories. Therefore, I would like to take the time to express my gratitude to everyone who has been seeing me and supporting me.

First and foremost, I would like to thank my Ph.D. advisor, Professor Tianyin Xu. I wouldn't have been able to reach this far were it not for Tianyin. I am thankful that Tianyin accepted me into the program back in 2020 (by digging my application out of the huge pile). As someone who directly went for a Ph.D. after undergraduate study, I lacked the correct mindset of a graduate student at that time and often fell short of his expectations. But Tianyin still constantly encouraged me—he helped me find my research interest in OS kernels and introduced me to those who later became my close collaborators throughout my Ph.D., for which I am extremely grateful. Besides research, Tianyin is also a life mentor and a personal friend. He helped me stay strong and continue pursuing what I believe through the difficult times during my journey. Honestly, I always feel I do not deserve such a good advisor.

Next, I would like to thank my co-advisor, Professor Dan Williams. I was introduced to Dan by Tianyin. We started the Rex project and have worked together ever since on all the projects discussed in this dissertation. Dan is always full of ideas and insights, which gradually influenced me and helped me learn to reason about problems in my research. Moreover, he taught me how to present my research ideas well in both paper writing and presentation. I am also grateful that Dan hosted me as a visitor student in the ROSA lab at Virginia Tech. I had a great time in Virginia and enjoyed the day-to-day discussions with Dan as well as the awesome students at ROSA.

I have had two wonderful internships at IBM Research, and I want to thank my mentor, Michael V. Le, my manager, Hani Jamjoom, and other colleagues I met: Salman Ahmed, Qiushi Wu, and Mimi Zohar. Mike had been my mentor for both internships and provided me with a superb internship experience. Like Dan, Mike has been my close collaborator ever since I did the first internship in 2022, and I am glad to see that our work has come to fruition in this dissertation. Mike is always optimistic and energetic, which also makes me feel energized by interacting with him. Hani is more like a life coach to me. He encouraged me to continue pursuing a research career path when I was in a state of self-denial. I still remember his "soul searching" advice, which I still use as my GitHub status today. Salman worked with me on both projects (i.e., Uno-kprobe and eKCFI) from the IBM internships and has been a close peer to me throughout the process. I truly enjoy our interactions. I met Qiushi during my first internship at IBM in the summer of 2022, when he was also an intern. Qiushi introduced me to LLVM-based static analysis and transformations. He helped me understand the basics of LLVM and referred me to the code of the artifact of his existing project. Qiushi later joined IBM Research as a full-time researcher during my second internship in 2023. I still remember all the fun we had while hiking and eating together in Flashing. Mimi is a kernel maintainer in our group. She taught me all the etiquette for sending patches to the Linux kernel and helped me shape up my first kernel patch, which I really appreciate. We also had many interesting discussions on Linux in general.

I also want to thank all the remaining members of my thesis committee: Professor Hubertus Franke, Professor Darko Marinov, and Professor Adam Bates. I had been working with Hubertus since my first Seccomp-eBPF project, and later, we worked on Rex together. I am grateful for all the wisdom he shared with me and his push for me to have a philosophical vision of kernel extensions in this dissertation, which made me think carefully about the meaning of my research. Although Darko never participated in any of the projects discussed in this dissertation, we have had many interactions over the years during our collaboration on non-thesis projects and general discussions. Darko often asks me really intriguing but easily overlooked questions about my projects, and I enjoyed the process of seeking the answers. Darko's practice of steering the project (e.g., the emphasis on automation) also influenced how I made progress in my own projects. The help from Adam on my Ph.D. journey is also significant—I took his operating system security class in my first year, which inspired me a lot and was extremely helpful for me to bootstrap myself for OS research (it is a pity that we were not able to collaborate on Seccomp-eBPF). I am very grateful for all the advice and suggestions Adam gave me during the thesis process.

Certainly, my Ph.D. life would be incomplete without all my labmates and friends. I would like to express my gratitude to all of my labmates: Xudong Sun, Siyuan Chai, Yinfang Chen, Jiawei (Tyler) Gu, Shuai Wang, Wentao Zhang, Minh Phan, Ayush Bansal, Zhizhen (Cathy) Cai, Cody Rivera, Jackson Clark, Yiming Su, Xuhao Luo, Shizhuo Zhang, Kaizhuo Yan, Neil Zhao, Jiyuan Zhang, Ruowen Qin, and Hao Lin. In particular, I want to thank Ruowen for constantly working with me on the Rex project, even after his graduation. Ruowen's strong technical skills gave a significant push to Rex—his contribution has far exceeded that of a regular second author, and the project would not have reached the current state were it not for him. Besides this, Ruowen is one of my closest friends during my Ph.D. Together, we discussed trending technologies, upstreamed kernel bug fixes, and watched Formula One—

all these memories will never fade. I also want to give special thanks to Xudong, Siyuan, Jiyuan, and Hao for our in-depth technical and research discussions, which have become an essential part of my research life. I appreciate YiFei Zhu for sharing his knowledge of the Linux kernel and Gentoo, which eventually made me a passionate kernel and system hacker. The students from the ROSA lab at Virginia Tech were very welcoming to me. I would like to thank Siddharth Chintamaneni, Milo Craun, Egor Lukiyanov, Raj Sahu, Sai Roop Somaraju, and LingXiang Wang—I truly enjoyed our technical discussions. I want to further thank my old friends from the University of California, San Diego: Zhidong Cao, Han Li, Jimou Li, Weifan Zhang, and Zihao Zhou. Thanks for keeping me positive over the years.

The Computer Science department also has a group of awesome staff that guided me through my Ph.D. process. I would like to give special thanks to Jennifer Comstock for her tremendous help with my thesis process, and Ruth Anders for her management of conference travels.

Lastly, I want to express my deepest gratitude to my parents. They have always been supportive of my decision to pursue a Ph.D. and has constantly encouraged me in my life. So long, thanks for everything you unconditionally give me—I love you two!

The research in this dissertation was supported in part by NSF grant CNS-1956007, CNS-2236966, an IBM-Illinois Discovery Accelerator Institute (IIDAI) grant, and a grant from Boeing Co.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

Kernel extensibility is an essential capability of modern Operating Systems (OSes). Kernel extensions allow users with diverse needs to customize the OS without adding complexity to the core kernel code or introducing disruptive kernel reboots.

In Linux, kernel extensibility has traditionally taken the form of loadable kernel modules. However, kernel modules are inherently *unsafe*—just like the kernel itself, modules are implemented in C, which provides no safety guarantees. Furthermore, after loading, module code executes directly in kernel space, which allows simple programming errors to directly crash the kernel. Acknowledging the problem, the Linux kernel recently incorporated Rust into kernel module implementations for its safety benefit [1]. Despite the support, there is no systematic support to ensure the safety of kernel modules—*unsafe* Rust code is allowed in kernel modules wherein checks to prevent errors are non-existent. Moreover, the vast, arbitrary interface exposed to kernel modules creates significant challenges in providing a safe Rust kernel abstraction to enforce safe Rust code [2, 3].

The interest in *safe* kernel extensibility has risen over the years [4, 5, 6, 7, 8]. One notable example is the eBPF extensions, which have gained significant traction and become the *de facto* kernel extensions [6, 9] on Linux. Core to eBPF's value proposition is a promise of *safety* of kernel extensions, enforced by the in-kernel verifier. The verifier statically analyzes extension programs in eBPF bytecode, compiled from high-level languages (C and Rust). It performs symbolic execution against every possible code path in the bytecode to check safety properties (e.g., memory safety, type safety, termination, etc). The kernel rejects any extension the verifier fails to verify. Today, eBPF extensions have evolved far beyond simple packet filters (its original use cases [10, 11]) and are increasingly being used to construct large, complex programs that customize storage [12, 13, 14, 15], networking [16, 17, 18], CPU scheduling [19, 20, 21], etc.

With the advancement of safe kernel extensibility, problems associated with safe kernel extensions have gradually emerged. This dissertation addresses three important problems: the lack of expressiveness, the overhead during execution, and the difficulty to program, in the practical use cases of modern safe kernel extensions, and explores the opportunities for new safe kernel extensibility designs opened up by these problems.

1.1 THE PROBLEMS

In this section, we introduce the three problems that arise in practical use cases of existing safe kernel extension frameworks: **expressiveness**, **efficiency**, and **programmability**.

1.1.1 Problem 1: Kernel Extensions Are Not Expressive

Despite the popularity of the modern eBPF extensions, Linux still does not employ kernel extension mechanisms that are expressive enough to support specific use cases, significantly hurting their efficacy. One representative example is programmable system call filtering.

Modern computer systems run a large variety of untrusted applications (e.g., in containers) 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. System call security policies are typically implemented in programs called *filters*. Filters are invoked at the entry point of every system call to decide whether to allow or deny each system call.

Seccomp (SECure COMPuting) [22] is the modern system filtering mechanism on Linux. Today, Seccomp is widely used by lightweight container and virtualization technologies such as Docker [23], gVisor [24], Firecraker [25, 26], LXC/LXD [27], Rkt [28], Singularity [29], Kubernetes [30], and Mesos Containerizer [31]. It is also used by system management services like `systemd` for user process sandboxing [32] and by Google's Sandboxed API project [33] for sandboxing C/C++ libraries. Hence, Seccomp provides "*the most important security isolation boundary*" [26].

Since Linux v3.5, Seccomp allows custom security policies to be written in the form of kernel extensions of *classic* BPF (cBPF) [34] in its filter mode. Filter mode enables application-specific security policies and results in the wide adoptions of Seccomp by different applications. Unfortunately, the expressiveness of cBPF is overly limited—security policies in Seccomp are mostly limited to *static* allow/deny lists of system call IDs and non-pointer argument values. 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). Recognizing the need for more expressive policies, Seccomp recently added a new feature known as Notifier [35], to support system call security policies implemented by trusted userspace agents found in early system call filtering techniques [36, 37]. Such technique, unfortunately, suffers from expensive user-kernel context switches, as well as other pitfalls such as time-of-check-to-time-of-use (TOCTTOU) based race conditions of argument values [38, 39, 40] compared with its kernel-extension-based counterparts, and is inadequate for complex system call filtering policies.

To address this lack of expressiveness of kernel extensions in the context of programmable system call filtering, this dissertation seeks to answer the following question: How can a kernel-extension-based framework that is both expressive and secure be constructed for programmable system call security?

1.1.2 Problem 2: Kernel Extensions Are Not Efficient

Safe kernel extensions are hindered by their suboptimal program execution. This dissertation discusses this problem in two contexts: (1) kernel extensions executing on top of kernel probes and (2) kernel-extension-based kernel control flow integrity.

Kernel Probes

A kernel probe is a kernel observability primitive that allows instrumentation of an arbitrary kernel instruction and executes user-defined "probe handlers." Contemporary operating systems, such as Linux, allow kernel extension programs to be used as kernel probe handlers. This combination provides a powerful yet safe functionality for use cases ranging from tracing [41, 42] to system monitoring [43, 44] and tuning [45], as well as dynamic patching [46].

Kernel probes are dynamic and not compiled into the kernel statically. To this end, kernel probes are usually implemented via traps. When a kernel probe is registered to a target instruction, the kernel rewrites the instruction into a trap instruction (e.g., `int3` on x86). When the execution reaches the target instruction (and therefore the trap instruction), a trap occurs, and execution is context-switched to the trap handler, at which point the kernel probe handler is executed and the original instruction is single-stepped. The execution eventually switches back to the original context to continue at the next instruction after the target.

The apparent problem with trap-based kernel probes is the excessive performance overhead resulting from the context switches. Our experiment measured a single kernel probe taking over *6,000* cycles on Linux. Such overhead significantly hinders the usefulness of kernel extension use cases. This dissertation, therefore, searches for an answer to the following question under the larger scope of kernel extension efficiency: How could a trapless kernel probe mechanism be built to support efficient kernel extension use cases?

Kernel Control Flow Integrity

Control flow integrity (CFI) [47] is an important technique for preventing control flow hijack attacks. CFI ensures the control transition points (i.e., indirect calls and jumps, and return instructions) in a program adhere to known valid targets. Given the critical position of operating system (OS) kernels in system security, CFI has naturally been applied to the kernel [48, 49, 50, 51], known as KCFI. It is supported in mobile OSes such as Android [52] and Apple's iOS [53], as well as server OSes like Windows [54]. The LLVM-based KCFI (LLVM-KCFI) for Linux has also been actively developed since v5.13 and was merged in v6.1 [55, 56].

However, existing KCFI is limited by its inflexibility due to the tight coupling of mechanism and policy. The *de facto* KCFI mechanism (LLVM-KCFI) for Linux relies on the compiler to statically instrument kernel code based on predefined policies (whether, where, and when to instrument CFI checks). After a KCFI-enforced kernel is deployed, it is hard to change the policies—doing so requires recompiling and rebooting the kernel, which can be disruptive and time-consuming in practice. It is also hard to support use cases that use KCFI for specific time duration, code regions or for specific processes.

In fact, LLVM-KCFI uses a static, coarse-grained policy that checks indirect control transfers based on function type matching. Such policies are permissive and imprecise, leaving the system vulnerable to advanced attackers [57, 58]. Unfortunately, with the current KCFI infrastructure, it is hard to adopt many advanced CFI policies such as those using advanced static analysis [49, 50, 59, 60, 61, 62]. As a result, KCFI policies are far behind user space CFI policies.

On the other hand, the eBPF kernel extension mechanism on Linux has the potential to form a foundation for overcoming the flexibility limitations of KCFI [63]. The programmable nature of eBPF enables the implementation of different KCFI policies; meanwhile, eBPF's dynamic nature as a kernel extension mechanism, where program loading and attaching does not require kernel reboots, can support flexible adjustments of KCFI policies at runtime. In fact, eBPF has been actively used in recent years to extend kernel security capabilities [64, 65, 66].

However, the existing eBPF infrastructure is fundamentally limited in supporting KCFI. There is no eBPF program type that can satisfy essential requirements of KCFI: (1) low execution overhead of eBPF policy programs, (2) scalable attachment of CFI checks as eBPF programs, (3) complete coverage on all indirect control flow transfers in kernel code. These limitations are demonstrated by tools that use eBPF to check control flows of handpicked instructions [43, 44], where the kprobe [67] program type is used to attach eBPF policy checks in kernel probes. Kprobe constantly uses traps to invoke eBPF checks, incurring expensive context switch overhead. As an eBPF program must be attached repetitively for every indirect control transfer instruction, these tools cannot scale to the entire kernel. Additionally, these tools only attach eBPF checks to the entries of kernel functions, having incomplete coverage.

This dissertation seeks to answer the second question regarding efficient kernel extension mechanisms: How can a KCFI infrastructure be designed and implemented on top of kernel extensions and offer the required efficiency, scalability, and coverage?

1.1.3 Problem 3: Kernel Extensions Are Hard to Program

The expanding popularity of eBPF as a safe kernel extension mechanism on Linux has ignited an industry around system-level capabilities from tracing and observability [68] all the way to networking [16, 17, 18], storage [12, 13, 15], and custom CPU scheduling and memory management policies [14, 19, 20, 21]. One of the key reasons that fueled its rapid adoption is the ability to execute custom code in kernel space, thereby extending kernel functionality with an unprecedented promise of *safety*. To this end, eBPF programs are compiled to a restricted bytecode, upon which the kernel performs *verification*: a form of symbolic execution to examine all possible program paths and guarantee properties, including memory safety, freedom from crashes, proper resource acquisition and release, and termination.

However, we observe that eBPF's static verifier introduces significant usability issues, making eBPF extensions hard to develop and maintain, especially for large, complex programs. For example, the eBPF verifier often incorrectly rejects *safe* extension code due to fundamental limitations of static verification and defects in the verifier implementation. When such false rejections happen, developers have no choice but to refactor or rewrite extension programs in ways that "please" the verifier. Such efforts range from breaking an extension program into multiple small ones, nudging compilers to generate verifier-friendly code, tweaking code to assist verification, etc (see Section 4.2). Some of the efforts also involve reinventing wheels and hacking eBPF bytecode, which creates significant cognitive overheads and makes maintenance difficult.

We argue that these usability issues are rooted in the gap between the programming language and the eBPF verifier, which we term the *language-verifier gap*. When writing eBPF programs, developers interact with the high-level language and naturally obey a *language contract* to align with the safety requirements of the language. The compiler also adheres to the language contract. Unfortunately, the verifier is not part of the language contract and has different expectations. As a result, verifier rejections may be surprising; the feedback (verifier log) is at the bytecode level and is hard to map to source code. As a result, developing eBPF extensions requires not only a deep knowledge of the high-level language and safety properties of kernel extensions but also a deep understanding of implementation details and quirks of the verifier.

Given the language-verifier gap in static-verification-based safe kernel extensions, this dissertation then tries to answer the question: How should a safe kernel extension framework that is also usable and easy to program be designed?

1.1.4 The Core of the Problems

We believe that the issues we identified on safe kernel extension mechanisms, as well as the questions we seek to answer in this dissertation, are centered around a deeper and more general research topic that concerns the design tradeoffs of safe kernel extension frameworks. Here, we divide the general design of safe kernel extension mechanisms into two essential axes.

The first axis regards the *safety* aspect. The goal is to ensure that the extension adheres to a set of well-defined safety rules when executing in the kernel to prevent undesirable consequences, such as crashing or kernel resource leaks. A variety of methods across the design spectrum to ensure safety have been explored by previous and contemporary frameworks: VINO [5] leverages software fault isolation (SFI) techniques to prevent extensions from misbehaving at runtime; while having no restriction on extension programming, it incurs excessive overhead at runtime. Frameworks such as SPIN [4] utilize safety from the programming languages, which pushes a range of safety checks to compile time and still allows expressive programming experience. However, language-based safety implies close coupling between the framework and a specific language. The family of BPF extensions on Linux (both classic and extended) employs static bytecode verification that completely keeps the safety checks out of runtime and avoids the tight coupling with a single language. Compared to eBPF, cBPF further employs a much simpler instruction set with a stateless design, which makes safety and even security easier to reason. The price of static bytecode verification, on the other hand, is the inevitable restriction on the expressiveness and programmability of extension programs.

The second axis concerns the *extensibility* aspect, which defines how extension programs should be attached and executed in the kernel. Invoking extension programs through indirect function calls makes the execution efficient but restricts the number of locations a program can be hooked without changing the kernel code, as the code for an explicit invocation must be present. In contrast, executing extensions from traps allows them to be attached almost anywhere but incurs significant performance overhead. While it is possible to optimize the traps and still allow programs to be attached anywhere (as we will show in this dissertation), such a design gives away simplicity.

1.2 THESIS STATEMENT

We argue that the issues we identified on existing safe kernel extension programs are a consequence of incorrect design tradeoffs, especially in the specific context they target. Therefore, our thesis statement is that:

> Resolving the right design tradeoffs can greatly improve the expressiveness, efficiency, and programmability of safe kernel extensions.

1.3 CONTRIBUTIONS OF THIS THESIS

This dissertation makes the following contributions towards the thesis statement on the expressiveness, efficiency, and programmability aspects of safe kernel extensions.

1.3.1 Making Kernel Extension Frameworks More Expressive

As discussed in Section 1.1.1, in certain use cases, such as system call filtering of Seccomp that uses cBPF, the kernel extension mechanisms leveraged do not match their demand for expressiveness. Such a problem is a direct result of the tradeoff between program safety/security and the expressiveness that is misaligned with the use case. cBPF inherited the limited expressiveness from its historical usage as packet filters, but it no longer matches the demand of system call filtering. Although the minimal language simplifies the reasoning of safety and security, which is arguably also important for system call filtering, we argue that with a careful design, we can build a framework that is expressive enough to support various essential policies while at the same time achieving the needed level of safety and security.

Programmable System Call Security with eBPF (Chapter 2)

To address the lack of expressiveness in Seccomp, we design and implement a new, programmable system call security framework that allows advanced system call filtering policies to be expressed, by leveraging eBPF for filter implementation. Naïvely opening Seccomp to eBPF is not a solution, as the existing eBPF infrastructure is not designed for the special context of Seccomp in both functionality and security aspects. To this end, we create a new *Seccomp-eBPF* program type, exposing, modifying, or creating 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. We further demonstrate that Seccomp-eBPF filters can be integrated into modern container runtimes and work with container features (e.g., checkpoint/restore in user space). Lastly, we show that our eBPF-based filtering can enhance existing policies (e.g., reducing the attack surface of the early execution phase by up to 55.4% for temporal specialization), and implement new policies that mitigate real-world vulnerabilities and reduce filter overheads.

Impact: We sent our Seccomp-eBPF patch set [69] to Linux upstream. Though it was not accepted eventually, the patch set created heated discussions on the mailing list about the use case and security of eBPF filters. We later gave a talk [70] at the Linux Plumbers Conference 2022. Our paper [71] is currently on arXiv.

1.3.2 Making Kernel Extension Execution More Efficient

Similar to the expressiveness problem, the suboptimal efficiency of kernel extension frameworks we identified in Section 1.1.2 is caused by the incorrect tradeoffs along the extensibility axis. The aim of the Linux kernel probes to provide a simple design that attaches extensions anywhere leads to the trap-based implementation and significantly degrades extension performance, especially when many extension programs are loaded. The same problem is also reflected in our KCFI use case—the existing attachment methods that may be used for KCFI do not balance performance, attachment coverage, and attachment scalability. We believe that the existing attachment methods of kernel extensions (i.e., eBPF) on Linux make suboptimal tradeoffs for existing use cases due to the emphasis on design simplicity and, therefore, also make them hard to support new uses. In this dissertation, we show that by trading off slightly more design complexity, we can build new attachment mechanisms that provide all the needed performance, scalability, and attachment coverage of our use cases.

Universally Trapless Kernel Probe Implementation (Section 3.1)

To address the efficiency challenges of extension programs on top of kernel probes, we aim to design a trapless kernel probing mechanism. Variable-length instruction set architectures such as x86 pose unique challenges to this goal, as it is difficult to dynamically patch kernel code without overflowing basic block boundaries. Executing the probed target remains another challenge, since certain instructions cannot be moved and executed elsewhere (e.g., on trampolines). Our insight on addressing these challenges is that by introducing strategically placed *nops*, thus slightly changing the code layout, we can utilize the extra space from the *nops* for code patching without overflowing basic blocks, and have an opportunity to execute the probed target in place without additional overheads. We design and implement our trapless kernel probing mechanism, Uno-kprobe, on Linux kprobe [67] and demonstrate that 96% of kernel code can be traplessly probed, with a 10x improvement of probe performance over the existing trap-based implementation.

Impact: We engaged with Linux kprobe maintainers and upstreamed improvements [72, 73] to the kprobe subsystem, which included fixes for bugs we found throughout the project, as well as further optimizations on the current kprobe implementation. Our work was published at USENIX ATC'24 [74].

Flexible and Easily Deployable KCFI with eBPF (Section 3.2)

As discussed in Section 1.1.2, the existing eBPF infrastructure is fundamentally limited in supporting a KCFI mechanism that is efficient, scalable, and has complete coverage. This limitation prompts us to design and implement eKCFI, a new framework that allows eBPF programs to be used for checking CFI at forward-edge control flows. eKCFI addresses the aforementioned challenges with a new compiler-based *nop* instrumentation similar to Uno-kprobe, thereby addressing the efficiency and coverage issue, and a new program attachment technique utilizing a global trampoline to avoid the additional scalability problem inherently associated with the existing kprobe infrastructure. We carefully designed and engineered the eKCFI infrastructure to prevent it from being tampered with, by introducing read-only eBPF maps and eliminating indirect calls on instrumentation code paths. We showcase that eKCFI not only offers flexibility and improved security with different KCFI policies from previous works [59, 75], but at the same time only incurs reasonable overhead.

Impact: Linux eBPF maintainers showed interest in the KCFI use case and our framework at the eBPF'23 workshop, where we published an extended abstract [63], and invited us for a talk [76] at the Linux Plumbers Conference 2023. The paper is currently under submission.

1.3.3 Making Kernel Extensions Easier to Program

Section 1.1.3 identifies the programmability and usability problem of the existing eBPF extensions on Linux. While static verification on bytecode allows safety checks to be performed before runtime and does not require programs to be implemented in a particular language, it inevitably creates the language-verifier gap, a mismatch between developers' expectation of program safety provided by a contract with the programming language, and the verifier's expectation. We argue that, by leveraging language-based safety, along with a lightweighted runtime, the need for static verification can be eliminated while still providing the desired safety guarantees for kernel extensions. Although such a design is bound to a particular programming language, we believe this is less of a problem than programmability and usability, and can be mitigated by using a widely adopted language.

Closing the Language-verifier Gap with Rex (Chapter 4)

We present Rex, a new kernel extension framework that closes the language-verifier gap and improves the usability of kernel extensions in terms of programming experience and maintainability. We design and build Rex on top of the inherent language-based safety from Rust to provide safety properties desired by kernel extensions, along with a lightweight extralingual runtime for properties unsuitable for static analysis, including safe exception handling, stack safety, and termination. We demonstrate that, with the absence of a static verifier, Rex effectively closes the language-verifier gap, as Rex programs no longer require the cumbersome and arcane workarounds against verification failures, and at the same time allow cleaner code and comparable performance to eBPF.

Impact: We published a workshop paper [77] at HotOS'23. The Rex paper [78] is on arXiv but also under submission. We used Rex to design three programming assignments (so-called "machine problems") on kernel tracing and networking for CS423 (Operating System Design) at UIUC.

CHAPTER 2: EXPRESSIVENESS

Expressiveness is an essential feature of any programable software framework. From general-purpose programming languages, to domain-specific languages, and to kernel extensions, such frameworks need to be expressive to fully support their use cases. However, we identified a mismatch between the expressiveness of kernel extension and its use case of programmable system call filtering in Seccomp [22] on Linux, where the design tradeoff overly favors the safety and security aspects of extension programs and sacrifices expressiveness. As the de-facto system call filtering framework on Linux, Seccomp allows users to specify a policy (a.k.a., a filter) that identifies the set of allowed and disallowed system calls of an application [79] for the sake of system security. Currently, the filter can only be implemented in the classic BPF (cBPF) [34] kernel extension. cBPF is an extremely constrained language in expressiveness, in which there are no mechanisms of keeping filter states or invoking kernel utility functions. Consequently, cBPF filters are generally limited to static allow/deny lists with only checks on system call numbers and non-pointer arguments. As such, cBPF kernel extensions cannot fully support its use case of programmable system call filtering, and many desirable and essential system call filtering features, which we shall identify and discuss in this chapter, are fundamentally not supported by Seccomp-cBPF. The focus of this chapter is to address the lack of expressiveness in system call filtering of the existing cBPF kernel extensions, and to construct a more programmable system call filtering framework leveraging more expressive kernel extension mechanisms. We show that with careful design, we can arrive at an optimal tradeoff that provides expressiveness to support desirable system call filtering features, while achieving the needed safety and security at the same time.

This chapter presents Seccomp-eBPF, a programmable system call filtering mechanism leveraging eBPF on top of Linux's Seccomp. 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 expressiveness, 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 naïvely opening the eBPF interface in Seccomp, as early Linux patches [69, 80, 81], 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 expressive for users to implement 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%.

This work 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`.

2.1 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.

2.1.1 Seccomp-cBPF

Seccomp currently relies on the *classic* BPF (cBPF) language for users to express system call filters as programs [79, 82]. cBPF has a very simple register-based instruction set, making the filter programs easy to verify. Due to the limited expressiveness, 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 [83].

2.1.2 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 [35]) to complement cBPF filters. Similar to early system call interposition frameworks [36, 37, 38, 84], 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

```go
// Without NoNewPrivileges, seccomp is a privileged operation,
// so we need to do this before dropping capabilities
if l.cfg.Config.Seccomp != nil && !l.cfg.NoNewPrivileges {
  seccomp.InitSeccomp(...)
}
// Drops the capabilities, sets the correct user and working dir,
// before executing the command inside the namespace
finalizeNamespace(...)
...
pdeath.Restore()
...
if unix.Getppid() != l.parentPid { ... }
...
unix.Write(fd, []byte("0"))
...
system.Exec(name, l.cfg.Args[0:], os.Env())
```

Figure 2.1: Container-launching code from runc

(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 [85]. For all of these reasons, Seccomp Notifier is inadequate for complex system call filtering policies.

2.2 ESSENTIAL SYSTEM CALL FILTERING FEATURES

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

2.2.1 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 (Section 2.6.2). 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 [86], a container runtime used by Docker and Kubernetes, shown in Figure 2.1:

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 [87, 88].

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 [89] 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 [89]. Moreover, prior work on IDS (Intrusion Detection Systems) has showed that an attack can be modeled by a sequence of system calls [90, 91, 92, 93, 94, 95]. Support for stateful filters that maintain sequences and state machines would enable a system call filtering framework to implement both SFIP and IDS enforcement.

Precise Temporal Specialization

Prior work [96, 97, 98] 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. [96] 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 2.5 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.

2.2.2 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 [79, 82]. Complex policies result in long lists of jumps and multiple filters (due to the size limit of a cBPF filter, see Section 2.1). Consequently, system call checking becomes expensive due to the need of iterating over long jump lists [79, 81, 99, 100, 101] and the overhead of indirect jumps caused by mitigations to speculative vulnerabilities (e.g., Retpoline) [83]. To reduce the overhead, multiple optimizations were proposed, such as dedicated Seccomp caches [100], skip-list search [102], and filter merging [83].

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 [100].

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 Section 2.4, 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.

2.2.3 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 [103, 104, 105, 106, 107]. 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.

2.2.4 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 [108, 109]. 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 [38, 39, 40], 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.

2.3 THREAT MODEL AND DESIGN GOALS

2.3.1 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` [110] 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.

2.3.2 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 Section 2.2. This not only requires a more 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 (Section 2.3.1). 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 (Section 2.3.1), our design must ensure no privilege escalation.

2.4 DESIGN

2.4.1 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 Section 2.2). eBPF is a fundamental redesign of the BPF infrastructure within the Linux kernel [111]. 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 [112], 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 2.2 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 expressive to support 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.

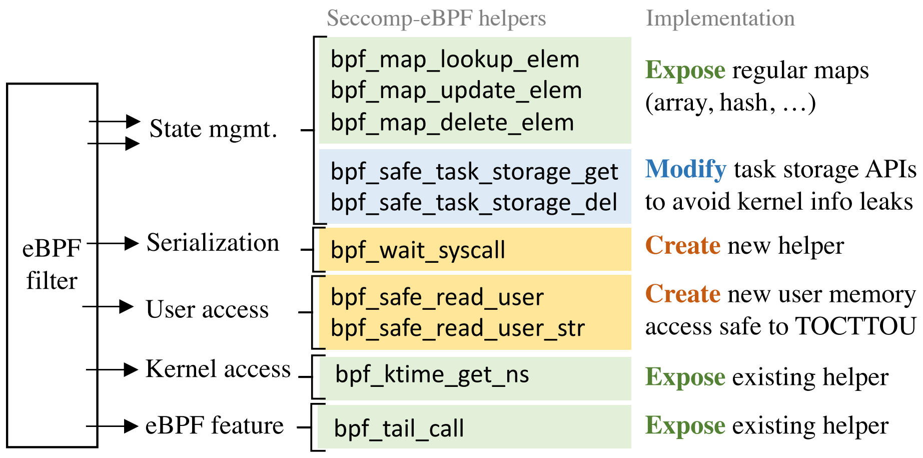


Figure 2.2: Features, helper interfaces, and their implementation of the Seccomp eBPF program type.

2.4.2 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 2.3.

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 (Section 2.4.3). 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 2.3 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.

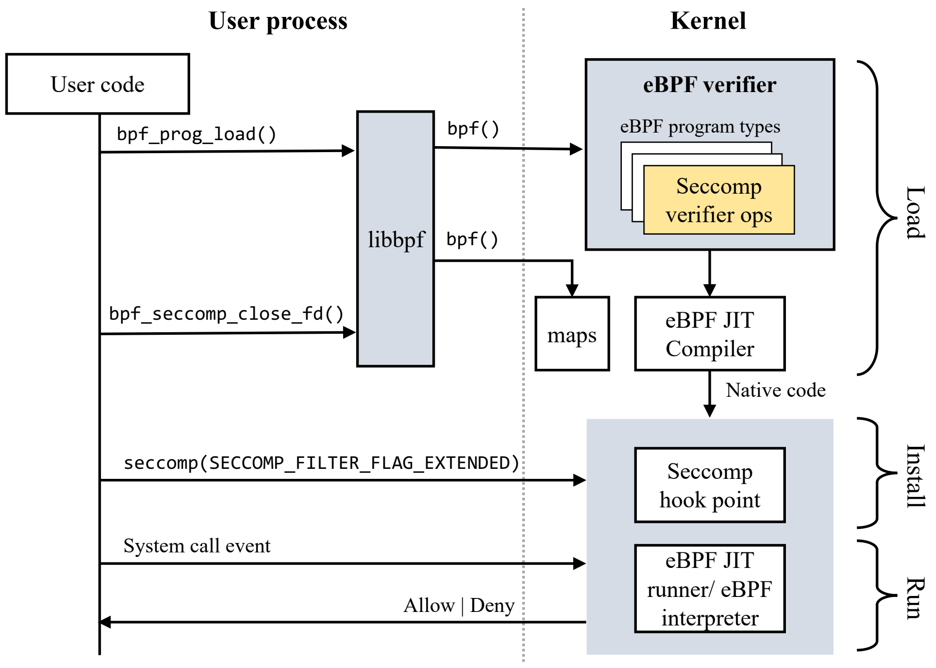


Figure 2.3: Workflow of a Seccomp-eBPF filter

2.4.3 Helper Functions

Figure 2.2 shows the helper functions we expose to 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 exists 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.

Task Storage

Maps are the basic enabler to statefulness (Section 2.2.1). 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_PRIVS`, see Section 2.3.1). 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 2.2)? 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.

Safe User Memory Access

Safely accessing user memory is an important feature for checking pointer-typed argument values, as discussed in Section 2.2.4. 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 [38, 105].

We follow the prior work on deep argument inspection [108, 113]. 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 [114, 115, 116]. We implement the first two solutions, as they can be done on all commodity hardware, and expose them with user-memory access helpers. Chapter A discusses our implementations.

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 [35] 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 option, `ptrace_scope` [117]. 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 [118]. 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 (Section 2.5). Therefore, our user-memory access helpers can handle page faults.

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 (Section 2.2.3). 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.

2.4.4 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

```c
struct sock_fprog prog = ...;
prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0);
// Load and install the filter in Seccomp
seccomp(SECCOMP_SET_MODE_FILTER, 0, &prog);
```

(a) cBPF: load and install in one step

```c
// Load the eBPF filter in bytecode
bpf_prog_load(filter_path, BPF_PROG_TYPE_SECCOMP, &obj, &fd);
prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0);
bpf_seccomp_close_fd(obj);
// Install eBPF filter in Seccomp
seccomp(SECCOMP_SET_MODE_FILTER, SECCOMP_FILTER_FLAG_EXTENDED, &fd);
```

(b) eBPF: load and install in two steps; verification invoked on load

Figure 2.4: Code snippet for installing Seccomp filters in (a) cBPF and (b) eBPF. `bpf_prog_load` is the libbpf function that wraps the `bpf()` system call; `bpf_seccomp_close_fd` closes all file descriptors (FDs) except the eBPF program FD (which is needed as a parameter for `seccomp()` and gets closed inside).

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 (cBPF "*is frozen* [119].").

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 2.4 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. However, the separation requires additional efforts to protect filters and maps against tampering (Section 2.4.5).

2.4.5 Tamper Protection

As a Seccomp-eBPF filter is verified and installed in two steps (Section 2.4.4), 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 closed by `seccomp` itself. The maps themselves are ref-counted by the eBPF filter program after eBPF verification; therefore, closing the map FDs does not remove the maps [120].

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 [121]) 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.

2.5 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 (cBPF programs do not need to sleep), e.g., for page fault handling when accessing user memory (Section 2.4.3). 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 cBPF 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 [101, 122]. It has been mentioned that Seccomp-eBPF filters are desired [123]. 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 [124], a fast OCI-compliant container runtime and the default container runtime of Podman [85]. The code can be found at [124]. Attaching an eBPF filter to a Podman container can be done with the following command:

```
podman --runtime /usr/local/bin/crun run
       --annotation run.oci.seccomp_ebpf_file=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. With it set, Seccomp-eBPF can only be used by processes with the `CAP_SYS_ADMIN` capability. Even with the configuration set, Seccomp-eBPF is still very useful for container runtimes and other management contexts (e.g., `init`) that run under the root privilege. For example, most of the container frameworks are not rootless.

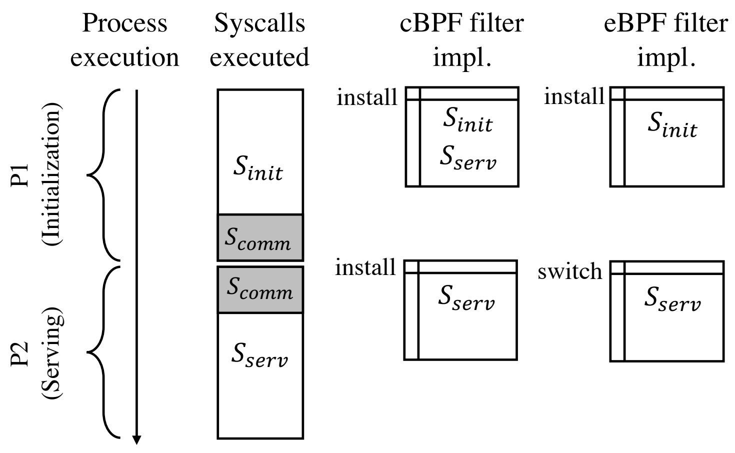


Figure 2.5: Two-phase temporal specialization implemented with cBPF and eBPF filters. S_{init} and S_{serv} refer to the set of system calls required by the initialization and serving phase, respectively; $S_{comm} = S_{init} \cap S_{serv}$

2.6 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 (Section 2.6.1). 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 2.2. We then evaluate the performance of eBPF filters with both micro and macro benchmarks. We also use eBPF filters to accelerate stateless checks (Section 2.6.3).

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.

2.6.1 Precise Temporal Specialization

We explained how eBPF filters could enhance the security of temporal system call specialization (Section 2.2). This section quantifies the security benefits of implementing temporal specialization in eBPF filters and comparing them with existing cBPF filter implementations [96]. We evaluate temporal specialization for two distinct phases—P1 (initialization) and P2 (Serving), as illustrated in Figure 2.5.

With cBPF filters, we install two filters: the first filter, installed at process startup, allows $S_{init} + S_{serv}$; the second filter only allows S_{serv} and is installed at the start of the serving phase (the location is provided by the application developer). Hence, cBPF filters cannot

```c
unsigned int phase;
SEC("seccomp")
int bpf_prog_ts(struct seccomp_data *ctx) {
  if (!phase && ctx->nr == -1) {
    phase = 1; // phase change
    return SECCOMP_RET_ALLOW;
  }
  if (!phase) { // only allow S_init
    switch (ctx->nr) {
    case SYS_alarm: return SECCOMP_RET_ALLOW;
    ...
    default: break;
    }
  } else { // only allow S_serv
    switch (ctx->nr) {
    case SYS_accept: return SECCOMP_RET_ALLOW;
    ...
    default: break;
    }
  }
  return SECCOMP_RET_ERRNO | EPERM;
}
```

Figure 2.6: eBPF temporal specialization filter

precisely implement temporal specialization, i.e., only allowing S_{init} for the initialization phase. As discussed in Section 2.2, 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.

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_{init} for the initialization phase and only S_{serv} for the serving phase. Hence, it addresses the limitations of cBPF filters. Figure 2.6 sketches this filter.

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.

We use the six server applications in the temporal specialization work [96]. We use S_{init} and S_{serv} of each application, provided by the research artifact [96], from which we derive S_{comm}. Table 2.1 shows the attack surface reduction of the initialization phase achieved by eBPF filters over the cBPF filters. The eBPF filters reduce the attack surface of the initialization phase by 33.6%–55.4% across the evaluated applications.

2.6.2 New Security Features

We present new security features enabled by Seccomp-eBPF filters and show that they can effectively mitigate real-world vulnerabilities (Table 2.2).

Table 2.1: The numbers of unique system calls in different phases of the evaluated applications

Application	$\lvert S_{init} \rvert$	$\lvert S_{serv} \rvert$	$\lvert S_{comm} \rvert$	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	53	135	44.4%

Table 2.2: Evaluated vulnerabilities and their patterns that can be effectively defended by Seccomp-eBPF filters.

Vulnerabilities	Pattern	Involved Syscall(s)	eBPF Filter Defense
CVE-2016-0728	Repeated syscalls	`keyctl`	Count limiting
CVE-2019-11487	Repeated syscalls	`io_submit`	Count limiting
CVE-2017-5123	Repeated syscalls	`waitid`	Count limiting
BusyBox Bug #9071	Syscall sequences	`socket` → `exec/mprotect`	Flow-integrity prot.
CVE-2018-18281	Raced syscalls	`mremap`, `ftruncate`	Serialization
CVE-2016-5195	Raced syscalls	`madvice`, `ptrace/write`	Serialization
CVE-2017-7533	Raced syscalls	`fsnotify`, `rename`	Serialization

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 (Section 2.2). 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 N, all subsequent invocations of the specific system call are denied.

Figure 2.7 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.

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 2.2.

Also, we implement *rate* limiting for frequency of specific system calls, using timer helpers (Figure 2.2).

```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;
}
```

Figure 2.7: eBPF filter to limit the number of calls to `keyctl`.

Flow Integrity Protection

We use Seccomp-eBPF filters to implement the kernel enforcement of system call flow-integrity protection (SFIP) [89] 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 [125] in the evaluation of the original SFIP work [89]. The exploits require a system call sequence of either `socket` → `execve` to execute shellcode, or `socket` → `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.

Serialization

We implement the eBPF based serializer (see Section 2.4.3) to mitigate the race-based vulnerabilities in Table 2.2. 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.

Table 2.3: Execution time of system calls with different types of filters and filter execution time.

	getppid (cycles)	**filter (cycles)**
No filter	244.18	0
cBPF filter (default)	493.06	214.19
cBPF filter (optimized)	329.47	68.68
eBPF filter	331.73	60.18
Constant-action bitmap [126]	297.60	0
Seccomp notifier	15045.05	59.29

2.6.3 Performance

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

Microbenchmark

Table 2.3 shows our microbenchmark results of different Seccomp filters, including the execution time (cycles) of the `getppid` system call and the filter programs. We use a policy that denies 245 system calls and allows the rest (`getppid` is allowed). To obtain reliable results, we fix the CPU frequency and disable Turbo boost.

We generate the two cBPF filters using libseccomp (v2.5.2), with the default option and with binary-tree optimization [99]. The latter generates a filter that sorts the system call ID in a binary tree. We implement the eBPF filter in C and use Clang (`-O2`) to compile the C code into eBPF bytecode. Clang optimizes switch-case statements into a binary tree so that reaching each case only takes $O(log(N))$ time, where N is the number of system calls. We also experiment with Seccomp constant-action bitmap [126], 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.

Table 2.4: Applications and benchmarks used in evaluation

Application	**Description**	**Benchmark**
HTTPD	Web server	ab [127] (40 clients)
NGINX	Web server	ab [127] (40 clients)
Lighttpd	Web server	ab [127] (40 clients)
Memcached	In-memory cache	memtier [128] (10 threads)
Redis	Key-value store	memtier [128] (10 threads)
Bind	DNS server	dnseval [129]

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 Section 2.6.1. 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 Section 2.6.1. We use official benchmark tools for each application (Table 2.4). All experiments are run 10 times and the average numbers are reported. Figure 2.8 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.

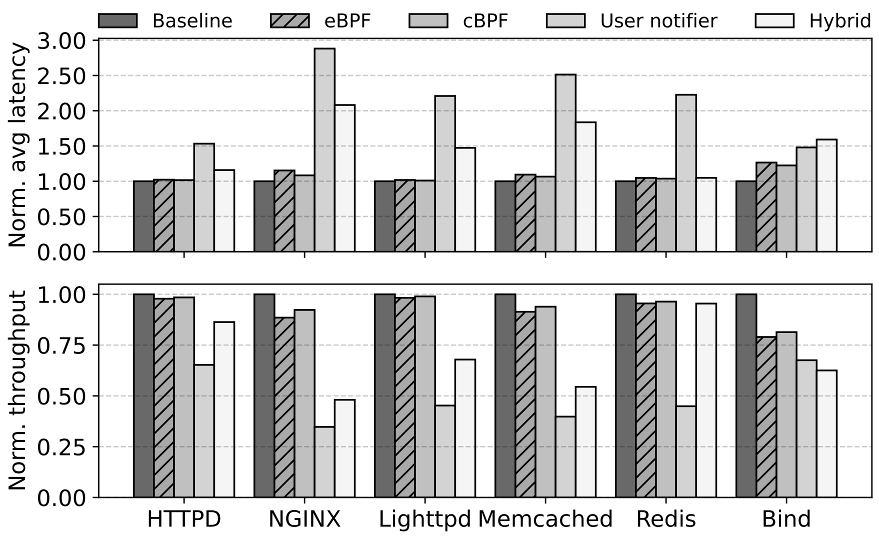


Figure 2.8: Avg. latency and throughput of the applications with different filters that implement temporal specialization (normalized to the baseline that disables Seccomp).

Accelerating Stateless Security Checks

Lastly, we use eBPF filters to implement the Draco Seccomp cache [100] and repeat the Draco evaluation using the applications and benchmarks in Table 2.4. 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 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 `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.

2.7 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 [130]. In fact, Seccomp(-eBPF) and LSM(-eBPF) are fundamentally different, and the enhanced expressiveness in Seccomp is essential.

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 sysem calls; 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-cBPF filters. The basic idea is to profile the system calls of the target application with static code analysis [96, 131, 132, 133], binary analysis [102, 133], or dynamic tracking [134, 135]. 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 [89] 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 [90, 91, 92, 93, 94, 95, 136].

Risks of Unprivileged eBPF

With bugs in Linux eBPF verifier and JIT compiler [137], 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 [137, 138, 139, 140, 141], and more broadly, safe and correct kernel code [142, 143, 144, 145].

Note that Seccomp-eBPF can be configured as a privileged feature at deployment (Section 2.5), if unprivileged eBPF is a concern. The privileged configuration can be used by many container runtimes and management services (e.g., `init`).

2.8 SUMMARY

This chapter addresses the lack of expressiveness of kernel extensions in the system call filtering use case to make system call security mechanisms more programmable. Our work makes one step towards empowering advanced system call security policies with better programmability of system call security mechanisms, namely Seccomp in Linux, that results from the enhancement. 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, and in turn, better expressiveness for kernel extensions.

CHAPTER 3: EFFICIENCY

This chapter identifies and seeks to address the efficiency limitations in existing kernel extension frameworks. Like other software systems, an OS kernel slowed down by the loaded kernel extensions is almost always suboptimal and undesirable. In this chapter, we discuss two cases where existing kernel extension frameworks are not efficient enough for their use cases. The problems in both cases are an artifact of the design tradeoff that emphasizes simplicity. We demonstrate in this chapter that by trading off slightly more design complexity, which we believe is desirable, we will be able to achieve the needed efficiency for our use cases.

We first discuss the case of kernel extensions executing on top of kernel probes (Section 3.1). Kernel probes implement a powerful observability primitive that can be utilized by kernel extensions for tracing [41, 42], system monitoring [43, 44], and other [45, 46] purposes. However, basic kernel probe implementations are implemented with traps, introducing prohibitive context switch overhead to its kernel extension use cases. Thus, in Section 3.1, we present a trap-free design of kernel probes that allows efficient execution of kernel extensions on top.

We later shift our focus toward the use case of kernel extensions in KCFI (Section 3.2). The inflexibility of existing KCFI mechanisms on Linux to take advantage of recent advances in analysis techniques prompts the desire to implement a more programmable KCFI mechanism using the eBPF kernel extensions. However, the existing eBPF infrastructure, such as fprobe-BPF [146] and kprobe-BPF [67], is limited in supporting efficient instrumentation of control flow transfer sites. In fact, the limitations are beyond just efficiency—the instrumentations are neither scalable nor able to cover all control flow transfers. We discuss the limitations in more detail in Section 3.2 and then present a new eBPF-based programmable KCFI mechanism that supports the required level of efficiency, scalability, and coverage.

3.1 UNIVERSALLY TRAPLESS KERNEL PROBES

Contemporary operating systems allow kernel extensions to be executed on top of kernel probes. The ability from kernel probes to instrument a running OS kernel is critical for not only debugging and event tracing [41, 42] but also for system security [43, 44], performance monitoring [45], and dynamic patching [46]. An efficient and fast kernel probing mechanism is key to enabling the use of these applications directly in the field on production systems.

A kernel probe allows users to dynamically instrument arbitrary kernel instructions to execute user-provided handlers—pre-handlers before executing the probed instruction and post-handlers after. To intercede on the kernel's control flow and invoke these handlers, typical kernel probe implementations rely on *traps*. When a probe is registered, the kernel makes a copy of the probed instruction and replaces it with a breakpoint instruction. When execution hits the breakpoint instruction, a trap occurs and the control is transferred to the probe subsystem, which executes the pre-handler, probed instruction, and post-handler before resuming normal execution at the instruction following the probe point. The obvious drawback of trap-based probes is the significant overhead due to the expensive context switches involved (more than *6,000* CPU cycles from our measurements).

To overcome the above drawback, a *trapless* approach is needed. The key idea is to replace the expensive traps with control-redirecting instructions like jump instructions. While a trapless approach can eliminate the overhead associated with the traps, it introduces a major challenge. Specifically, for variable-length instruction set architectures like x86, a jump instruction can be longer than the probed instruction, thereby overwriting multiple instructions. This can cause the jump instruction to span basic block boundaries and cause execution failure. This challenge can limit where trapless probes can be used, thus how many instructions can be trapless.

Whether using a trap-based or trapless approach, another challenge with implementing a kernel probing mechanism is how to execute the copy of the probed instruction efficiently. One typical choice is to execute the copied instruction directly on the processor. However, this direct execution does not work for some sets of instructions such as those related to the instruction pointer, e.g., calls and jumps. These instructions must be emulated, which is slow and adds complexity in terms of implementing and maintaining the emulation code.

In this section, we present the design and implementation of a universally fast (*trapless* probing on all probe-able code) kernel probing mechanism that requires no runtime code emulation. Our design allows probe handlers to be executed synchronously in the same context that triggered the probe and thereby avoids expensive context switches. To achieve this design, we rely on a key insight—by strategically inserting `nop`s into the kernel code, thus slightly changing the code layout, we can overcome the above two challenges. Specifically, for probe locations that straddle basic block boundaries, an inserted `nop` can ensure the jump instruction resides in one basic block; for instructions that require code emulation, placing a `nop` before such instructions allows a probe to be attached by overwriting the `nop` with a call to the kernel probe handlers, thereby allowing the probed instruction to be executed in place with no copy or emulation. Most importantly, in our approach, the locations to place the `nop`s can be automatically identified and kept minimal, thus reducing the impact on normal kernel operations when no probe is installed.

To demonstrate the efficacy of our approach, we apply our design to Linux kprobe [67] on x86, a widely used kernel probing mechanism and architecture. Kprobe has been transformed over the years from a purely trap-based probe mechanism to utilizing a trapless approach [46]. However, despite persistent optimization efforts, due to many fundamental limitations stemming from not being able to assume or change certain properties/layout of the kernel code at runtime and thereby overcoming the aforementioned challenges, kprobe is far from being universally trapless. Exacerbating the problem, existing kprobe optimizations are often applied in an ad hoc manner, resulting in many instructions being unoptimized, even if it is technically possible, oftentimes due to the sheer complexity of the implementation. We discovered that the existing kprobe is only optimized for ~80% of kernel instructions, leaving the remaining probe-able kernel code to suffer the severe penalties of up to two traps when probed.

Our universally trapless kprobe implementation consists of a new transformation pass in the LLVM x86 backend to identify the locations where a `nop` is needed and perform the insertion. To allow kprobes to be optimized using the inserted `nop`s, we implemented kernel support for efficient, scalable trampolines and runtime instruction rewriting.

We evaluate both the performance and optimization coverage of our design. Our kprobe implementation achieves a speedup of up to 3x for kprobe-based KCFI enforcements on LEBench [147] over the original kprobe, with the single-probe performance increased by a factor of 10x. Our kprobe implementation optimizes all the kernel instructions that can be optimized at compile time and brings the total instructions optimizable in the kernel to 96%. In fact, even more performance improvement can be potentially achieved (we prioritized compatibility and non-disruptive changes to accommodate the current optimizations of Linux kprobe).

In summary, this work makes the following contributions:

- We present a fast and universal trapless kernel probe design;
- We identify fundamental limitations of existing Linux kprobe optimization techniques;
- We implement our design on top of Linux kprobe and show the efficacy of the approach.

3.1.1 Design

The design of our trapless kernel probe mechanism has two main goals: 1) the kernel probe should be fast when applied to any kernel instructions without using expensive traps (e.g., `int3` exceptions) and 2) the mechanism should be simple to not increase implementation and maintenance complexity of kernel code and lightweight to not incur significant overhead. Since a kernel probe is inserted at runtime when the code layout is fixed, there are two main challenges with implementing a universally trapless kernel probe mechanism: 1) how to ensure the inserted jump instruction does not span basic block boundaries – blindly rewriting these instructions would overflow into the next basic block and corrupt branch targets, and 2) how to allow instructions that would normally require emulation in a typical kernel probe implementation to be directly executed. In this section, we discuss a clean-slate design of a universal trapless kernel probe mechanism before describing how we apply our design to Linux kprobe (§ 3.1.2).

A Baseline Solution

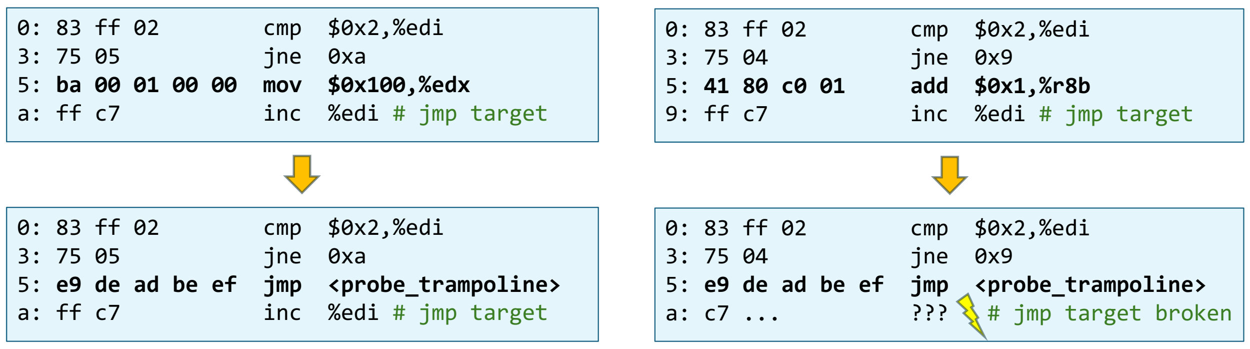
Our key insight for resolving the first challenge is to use the insertion of `nop`s to change the code layout of the kernel. The `nop`s inserted at compile time can provide space and be used as anchor points for inserting probes at runtime without tampering with regular kernel instructions.

The first basic incarnation of this approach is to insert a 5-byte `nop` instruction before *every* kernel instruction. In this way, probing a specific instruction works by rewriting the preceding `nop` into a same-sized relative `call` that redirects the control flow onto a global trampoline. The trampoline first saves register contexts on the stack. This prevents the user-defined handler from overwriting the current execution context. Next, the trampoline invokes the user-defined handler to perform actual probing. Finally, after the handler finishes, the trampoline pops the register contexts and returns. The returned control flow would land on the next instruction after the `call`, which is exactly the probing target. Execution can then continue with the correct context.

While this approach ensures the absence of traps, and thus fast kernel probes, inserting a `nop` before every instruction introduces significant overhead when the kernel probes are not used. Our evaluation shows a 75% slowdown for LEBench running on Linux v6.3.6 (§ 3.1.3).

Minimal `nops` Design

Clearly, improving the baseline would require *strategically* placing `nop`s. Fortunately, we observe that the vast majority of the `nop` instructions in the baseline can be omitted. Specifically, if there is enough space before a branch target, there is a potential opportunity of rewriting a relative `jmp` to the trampoline in place [46]. As illustrated in Figure 3.1a, the `mov` instruction being probed can be rewritten into a relative `jmp` to the probe trampoline,



```
0: 83 ff 02         cmp  $0x2,%edi
3: 75 05            jne  0xa
5: ba 00 01 00 00   mov  $0x100,%edx
a: ff c7            inc  %edi # jmp target
```

```
0: 83 ff 02         cmp  $0x2,%edi
3: 75 05            jne  0xa
5: e9 de ad be ef   jmp  <probe_trampoline>
a: ff c7            inc  %edi # jmp target
```

(a) Case where there is enough space before a jump target for the probed instruction.

```
0: 83 ff 02         cmp  $0x2,%edi
3: 75 04            jne  0x9
5: 41 80 c0 01      add  $0x1,%r8b
9: ff c7            inc  %edi # jmp target
```

```
0: 83 ff 02         cmp  $0x2,%edi
3: 75 04            jne  0x9
5: e9 de ad be ef   jmp  <probe_trampoline>
a: c7 ...           ???   # jmp target broken
```

(b) Case where there is not enough space before a jump target for the probed instruction.

Figure 3.1: Space contrains of rewriting an instruction into a jump instruction.

without tempering the branch target after it. Doing this rewriting requires us to copy the target instruction and any other instruction that could be overwritten by the 5-byte relative `jmp` to a temporary copy buffer. After the trampoline invokes the user-supplied handler and restores the register context, it executes the instructions saved in the copy buffer and performs another relative jump back to the normal execution path. This implementation requires the copied instructions to be executed out-of-line (in the buffer), not at the original code location.

Only in the cases where in-place replacement of a `jmp` instruction is not possible, we apply the `nop`-based design. Such cases can be broadly classified into two categories:

- instructions that cannot be executed out-of-line.
- instructions that do not have enough space without overwriting a jump target (i.e., start of a basic block).

The set of instructions that cannot be executed out-of-line includes instructions for which their text address matters for execution. These groups of instructions include `call` instructions and also instructions that may trigger page faults. The `call` instruction belongs to this group because the callee may be expecting a specific caller (e.g., via `__builtin_return_address`) and, therefore, cannot be executed directly on the copy buffer. Otherwise, the return address obtained would point to the copy buffer rather than the original address. On the other hand, certain instructions, e.g., userspace access instructions, could trigger a page fault if an invalid address is being accessed. Linux defines short "fixup code" snippets that implement the fault-recovery logic to allow execution to continue for such instructions [148]. The mapping between the faulting instruction and its fixup code is stored in a special exception table section of the kernel image. During a page fault, the page fault handler uses the address of

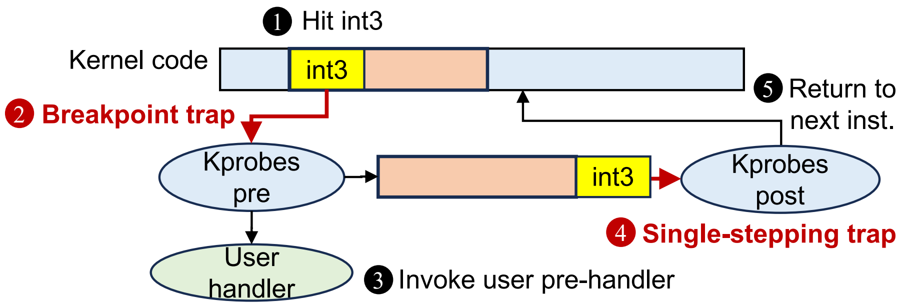


Figure 3.2: A kernel probe takes two traps (steps 2 and 4).

the faulting instruction to search for the corresponding fixup code in the exception table. This effectively prevents the instruction from being executed out-of-line in the copy buffer, since there is not an entry in the exception table that corresponds to the address of the faulting copy. In these situations, the inserted `nop` will still make it possible to probe such an instruction without the need for a trap.

For instructions at the end of a basic block, the rewriting technique may not be viable due to the possibility of overflowing into the next basic block when the instruction is not long enough (as shown in Figure 3.1b). Therefore, `nop`s are still needed under these situations since they allocate space to safely place a `call` for efficient probes that avoid the traps.

3.1.2 Implementation

We discuss the application of our approach to kprobe in Linux. We start by describing the current kprobe implementation with current optimization techniques. We then highlight the limitations of kprobe in achieving universally trapless kernel probing. Finally, we discuss how our approach can help overcome those limitations and present the details of a working implementation. Our aim is to have an implementation that minimizes changes to the kernel code, be fully compatible with kprobe, and ensure the changes are transparent to users.

Linux Kprobe

Without optimizations, a trap-based kprobe on x86-64 uses two traps (Figure 3.2). When a kprobe is registered, the kernel makes a copy of the probed instruction (to avoid race conditions) and replaces the first byte of the probed instruction with a breakpoint instruction, i.e., `int3` on x86-64. A second `int3` is appended at the end of the copied instruction.

When the execution hits the breakpoint instruction, a trap occurs, the CPU's registers

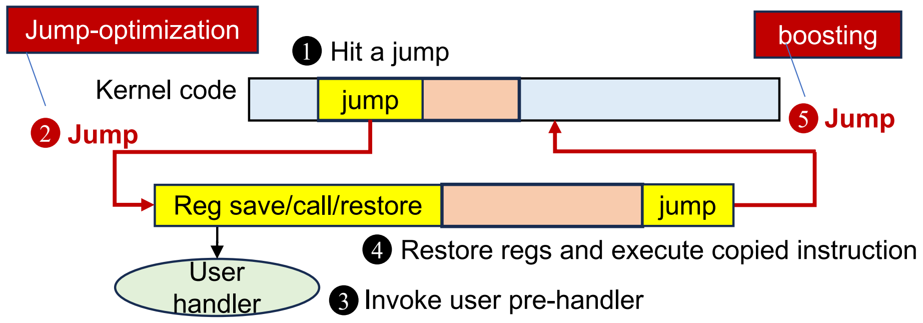


Figure 3.3: Existing kprobe optimizations known as boosting and jump-optimization.

are saved, and the control is transferred to the kprobe subsystem, which executes a user-provided pre-handler that is associated with the kprobe. Next, kprobe single-steps the copy of the probed instruction to ensure the kernel regains control to execute any user-installed post-handler associated with that kprobe. The single-step is implemented by the second `int3` trap after the copy. (known as "`int3`" single-step [149]) [1]. Execution then continues with the instruction following the probe point.

Trap-based kprobes suffer greatly from the expensive context switch overheads. Our measurement shows that such a kprobe consumes more than 6,000 CPU cycles.

Existing Kprobe Optimizations. Since trap-based kprobes suffer greatly from expensive context switches, optimizations have been developed to replace expensive traps with jump instructions [46] (Figure 3.3). In Linux, kprobe currently employs two optimizations, named *boosting* and *jump-optimization* for the two traps, respectively.

Boosting aims to replace the single-step trap with a jump instruction. The insight is that if a kprobe does not have a post-handler, then single-stepping may not be needed. So, for a kprobe without a post-handler, the boosting optimization adds a jump which jumps back to the next instruction, replacing the single-step trap. In this way, the kernel executes the copied instruction and the jump. Note that, unlike a pre-handler, a post-handler is not commonly used.

Boosting is limited in scope since it cannot handle instructions that change the instruction pointer register (`rip`), e.g., `call`, and the instructions which may require exception fixups. Instructions like `call` need to be emulated, instead of being executed out-of-line, because the return address pushed onto the stack needs to be corrected.

Jump-optimization builds upon boosting, aiming to replace the breakpoint trap with a five-byte relative jump that redirects the control flow to a pre-allocated trampoline. The

[1] The kernel uses `int3` instead of a Trap Flag (TF) to avoid implementation complexity of handling the side effect of using `iret` (interrupt return) [149]

Table 3.1: The number (percentage) of instructions in Linux (v6.3.6) that are not optimized by kprobes. There are in total 6,632,661 kprobe-able instructions.

Instruction	**Count (Percentage)**
Non-boostable	949,396 (14.31%)
Non-jump-optmizable (boostable)	445,037 (6.71%)
Total	1,394,433 (21.02%)

trampoline calls into the kprobe pre-handler and then jumps back to the next instruction. In this way, kprobe invokes the pre-handler synchronously on the same execution context and avoids the expensive context switch.

Note that jump-optimization may need to overwrite and copy multiple instructions, instead of only the probed instruction in the original kprobe (Section 3.1.2), because jump, as a five-byte instruction, may be longer than the probed instruction. The unoptimized kprobe with `int3` does not encounter this problem because `int3` is a one-byte instruction. Kprobe implements an x86 instruction decoder to recognize the instruction boundary.

Like boosting, jump-optimization is also limited. It suffers from the same basic block spanning issues described in § 3.1.1. The current kprobe implementation handles this limitation conservatively, because it cannot reason about basic block boundaries (i.e., branch targets) at runtime. Therefore, in addition to ensuring no near jump to the region of the jump instruction, kprobe also refuses to optimize *any* instructions in a function that contains indirect jumps. Also, the implementation of jump-optimization depends on boosting; so, non-boostable instructions cannot be jump-optimized.

Kprobe Limitations The kprobe optimizations are not only limited in scope but also applied to kernel instructions in an *ad hoc* way. Table 3.1 shows the amount of instructions that cannot be optimized by either boosting or jump-optimization in Linux (v6.3.6).[2] In total, 21.0% of the instructions in Linux cannot be fully optimized, i.e., attaching a kprobe on around one-fifth of the kernel text needs to pay for the context switch overhead raised by breakpoint exceptions. Among them, the majority are non-boostable (and thus cannot be jump-optimized).

In addition to the aforementioned limitations, the process of applying the two optimizations to the trap-based implementation (Section 3.1.2) is often ad hoc. Linux adopts a strategy of applying optimizations based on a few types of instructions that are deemed to be safe. Table 3.2 lists the types of instructions that are not non-boostable in Linux. For example, all Group 2, 3, and 4 instructions are deemed as non-optimizable in the kernel, while

[2]There are ~6K instructions that are not probe-able such as trap instructions like ud2 and int3 and functions labeled as non-traceable.

Table 3.2: Instructions that cannot be boosted (thus not jump-optimized in the current kprobe implementation).

1. `rip`-changing instructions, e.g. jumps and calls
2. Instructions that may trigger exceptions (e.g. pages faults)
3. Instructions that override address size or code segment
4. x86 instructions group 2/3/4/5 with reserved opcodes.
5. Instructions with 3-byte opcodes

some of them can clearly be optimized (e.g., `inc`/`dec` for increments/decrements). In fact, the dependency of the two optimizations is also an implementation artifact; in principle, the two optimizations are independent.

Universally Fast Kprobe

Instead of continuing to manually apply existing optimizations to more types of instructions and special cases, we seek a principled, universal implementation using our approach. Our implementation consists of a compiler transformation that selectively inserts `nop`s and kernel support for efficient trampolines and instruction rewriting. The former takes 549 lines of C++ code and the latter takes 298 lines of C code. We build on top of Linux v6.3.6.

Compiler Transformation We develop a compiler transformation to identify the minimal set of locations where `nop`s are needed to enable trapless kprobes. The transformation is implemented on LLVM as a `MachineFunctionPass` that works at the Machine IR (MIR) level [150] in the code-generation backend. We chose MIR instead of the generic LLVM IR because MIR closely models native code, which makes it easy to check whether an instruction can be optimized by kprobe at runtime.

The transformation takes two iterations and operates on each kernel function; it goes through each instruction in the function and identifies instructions that are not boosted or jump-optimized by Linux:

- If the instruction cannot be boosted, a `nop` instruction is inserted before that instruction.
- If the instruction can be boosted but not jump-optimized (which means that inserting a jump would overflow into another basic block), a `nop` instruction is inserted before the last instruction of the basic block.

Table 3.2 shows the categories of non-boostable instructions. For each non-boostable instruction category that can be identified by opcode, the compiler pass inserts a `nop` before the instruction.

Inserting a `nop` is straightforward in most cases. One special case is to handle *terminator instructions* of basic blocks, i.e., instructions at the end of the basic block that direct the control flow to the successor basic blocks. Since such instructions modify instruction pointers, they are almost always not boostable. The LLVM MIR implements the semantics similar to native assembly code and permits basic blocks to have multiple terminators (e.g., a basic block may have a conditional jump (`jcc`) followed by a direct jump (`jmp`)—the direct jump is executed when the conditional jump is not taken). The LLVM backend requires no non-terminator instructions between terminators. Therefore, it is invalid to insert `nops` between terminators, which is required for non-boostable terminators. To handle this case, our compiler pass splits the basic blocks with multiple terminators into multiple basic blocks, each of which has one terminator. The transformation maintains the original control flow, and allows the insertion of a `nop` in front of each terminator if needed.

The only non-boostable category that cannot be identified by opcode directly is the instructions that may trigger exceptions (e.g., page faults). Our design covers these instructions, but our current implementation does not handle them because the kernel exception table with the actual kernel address is not available until after linking.

For instructions that can be boosted but cannot be jump-optimized, our compiler pass aims to enable jump-optimization to avoid the breakpoint trap. The reason that jump-optimization is not applicable is due to the lack of space to place the five-byte jump instruction.

The problem is straightforward to fix at compile time, because basic blocks are explicit and branches can only jump to the entries of basic blocks. Therefore, the problem is reduced to handling the case when rewriting a jump would overflow to the next basic block. We address this case by inserting a `nop` before the last instruction of each basic block. Note that basic block terminators are handled by the prior iteration.

Kernel Support We keep the kprobe interface for probe registration. If the kprobe being registered is not jump-optimizable, our implementation will optimize it onto a preceding `nop` (if exists). This leverages the kernel text-patching interface to replace the 5-byte `nop` instruction with a 5-byte relative `call` instruction that redirects the control flow onto our trampoline.

In kprobe, the pre-handlers expect to receive the registers (`pt_regs`) of the context that triggered the probe. In trap-based kprobes (Section 3.1.2), registers are automatically saved by the processor. For jump-optimized kprobes that do not use a trap, their trampolines explicitly save the register context and invoke the user handlers with the saved context.

Currently, the kprobe jump-optimizer creates one trampoline for each instruction a jump-

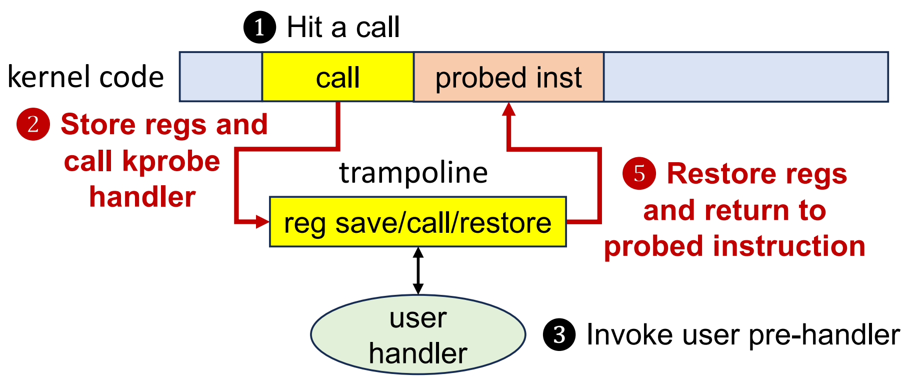

❶ Hit a call
kernel code
call
probed inst
❷ Store regs and call kprobe handler
trampoline
reg save/call/restore
❺ Restore regs and return to probed instruction
user handler
❸ Invoke user pre-handler

Figure 3.4: Optimized kprobe workflow by rewriting the `nop` into a `call`.

optimized kprobe is attached. The jump-optimizer copies a pre-defined trampoline "template" and fills it with information specific to that probe. For example, the kernel copies the instructions overwritten by the `jmp` to the end of the trampoline. Such a design is needed because the instructions cannot be directly executed in place after being overwritten by the `jmp` – they must be copied to a trampoline customized for that kprobe. At the same time, since jump-optimization uses a relative `jmp` to redirect the control flow to the kprobe pre-handlers, the jump-back address needs to be hardcoded into the trampoline in order to resume the normal execution after probing.

However, creating one trampoline for every probed instruction is not only complicated but also does not scale well. We address this issue by designing a single, global trampoline for all the kprobes that are optimized with the `nop` (Section 3.1.1), because the probed instructions can be executed in place.

Similar to the trampolines of jump-optimization, our trampoline pushes the registers onto the kernel stack with specific ordering so that the pushed values form a `pt_regs` struct which can be directly used by user handlers. A challenge for our one-trampoline design is to handle the instruction pointer register (`rip`). The pre-handler expects the `rip` to be the address of the probed instruction. However, the `rip` changes at each instruction and cannot be trivially pushed onto the stack. Unlike the per-instruction trampoline, the address of the probed instruction is not encoded on the global trampoline.

We solve the problem by rewriting our 5-byte `nop` into a 5-byte relative `call` instruction that calls onto the trampoline, instead of the relative `jmp` used by jump-optimization. The `call` instruction automatically pushes the return address onto the stack. This address is always the address of the next instruction after the `nop`, which is exactly the probed instruction and the `rip` value the handler expects. In this way, the trampoline can just read the value of the expected `rip` from the stack and store it in the `pt_regs`. The use of the

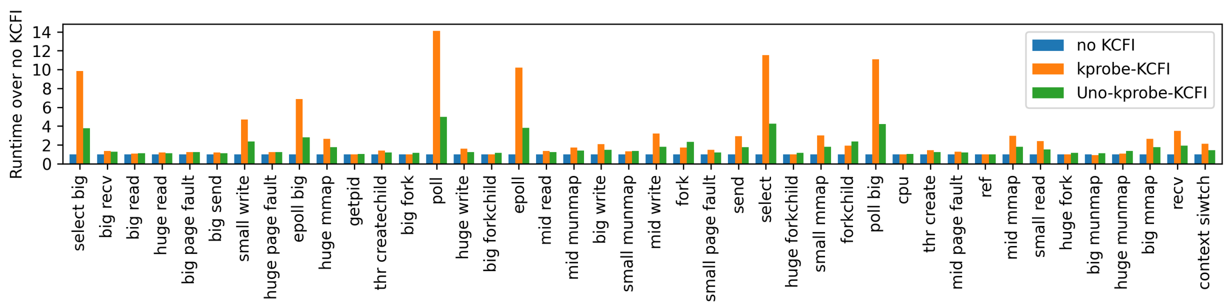


Figure 3.5: Runtime overhead of different KCFI policies when executing LEBench, with vanilla kprobe in Linux and Uno-kprobe. Value reported is based on median runtime of each benchmark.

`call` instruction also allows us to jump back to normal execution as a `ret` will be enough. The resulting workflow of our optimized kprobe is shown in Figure 3.4.

We implement our trampoline in x86-64 assembly. The finished trampoline after compilation takes a constant 96 bytes in the kernel, a significant improvement over the linear memory complexity incurred by jump-optimization.

3.1.3 Evaluation

We evaluate our trapless kprobe mechanism, denoted as Uno-kprobe, on its optimization coverage, probe performance, and `nop` overhead. All measurements are performed on a bare metal server with an 8-core Intel Xeon E-2174G CPU with 32GB of memory on a 1Gb/s network.

Trapless Kprobe Coverage

We assess the amount of kernel code that can utilize our trapless Uno-kprobe. Under the original kprobe implementation, the classes of kernel instructions that are not optimizable include instructions that cannot be boosted to execute out-of-line (e.g., `rip`-modifying instructions) and instructions at the end of basic blocks where jump-optimization could overwrite branch targets. Both classes can now be optimized with Uno-kprobe. Table 3.3 shows the amount of kernel code that can be optimized under the original kprobe and our Uno-kprobe. Our approach can optimize all currently unoptimized instructions that can be identified at compile time, bringing the total instructions optimizable in the kernel from 79% to 96%. The remaining non-optimizable instructions include instructions that could trigger page faults and instructions from inline-assembly blocks as well as assembly files

Table 3.3: Kernel code trapless-probe coverage of Uno-kprobe.

Total instructions	**Vanilla Kprobe**	**Uno-kprobe**
6.63M	79% (5.24M)	96% (6.38M)

Table 3.4: Latency (in cycles) of invoking an empty handler using vanilla kprobe and Uno-kprobe.

	Vanilla Kprobe	**Uno-kprobe**
No optimization	6235 ± 817	612 ± 407
Boost-only	2625 ± 2459	562 ± 369

(these instructions cannot be trivially processed in the LLVM code generator).

Performance

We first evaluate the performance of Uno-kprobeon a microbenchmark. Our experiment consists of a single kprobe with an empty handler attached to an instruction and measures the total latency of the kprobe and instruction. We measure the latency using both the original kprobe implementation and Uno-kprobeand on both a boostable but not jump-optimizable `nop` (located at the end of a basic block) and a non-boostable `shr`. The results are shown in Table 3.4. Uno-kprobe is about 10x faster than the existing kprobe on a non-boostable instruction (i.e., with two traps) and 5x faster than that on a boostable but not jump-optimizable instruction (i.e., with one trap) as both of these probe sites are fully trapless with Uno-kprobe. For existing kprobes on a non-boostable instruction, we found that the majority of the performance overhead (86.8%) comes from the traps and related context switches.

We then evaluate Uno-kprobeperformance on applications utilizing kprobes. Specifically, we measure the performance of the LEBench [147] benchmark with a kprobe on all indirect calls in the kernel that resembles a kernel CFI (KCFI) use case. Figure 3.5 shows the overhead of LEBench when KCFI is enabled using different techniques for invoking CFI handlers. As can be seen, using the original kprobe results in the highest overhead, with some system calls having overhead up to 3x compared with our trapless kprobes. On average, Uno-kprobeachieves a speedup of 1.4x across all LEBench. Kprobe has been recognized by prior work to perform poorly in use cases that require a high rate of invocation [44, 63]. Uno-kprobemakes it feasible to use kprobe for such an application.

Lastly, we evaluate the overhead introduced by our inserted `nop`s when kprobes are not used. We measure the performance of the LEBench under different `nop` insertion strategies: vanilla kernel (no `nop`), `nops` before non-optimizable instructions, and `nops` before every instruction. We found that inserting `nops` before every instruction yields a rather large overhead, 30% on average. Uno-kprobe, in contrast, incurs an overhead of 10% on average on LEBench when kprobes are not registered and achieves a performance advantage of up to 1.5x (`mid mmap`) over inserting `nops` before every instruction. This may be acceptable for applications that heavily rely on the probing functionality and already must incur the probing overhead, given the performance boost (up to 10x) at all probe-able locations that can be achieved with Uno-kprobe.

3.1.4 Summary

This section presented a fast and universal trapless kernel probe design based on strategically placed `nops`, in order to solve the inefficiency of kernel extensions executing in kernel probes, and further support our implementation of an eBPF-based KCFI framework. We demonstrated the feasibility by implementing our design on top of the Linux kprobe subsystem, which in some cases still relies on expensive traps. We show that the performance of kernel probes can be effectively improved through this general design. To make our work more practical, we have also engaged with the Linux kernel community and have upstreamed patches that contain optimizations [73] as well as bug fixes [72] for the current kprobe implementation. For future work, we are aiming to further optimize the Linux kprobe with trapless probes and make it more complete.

3.2 FLEXIBLE AND EASILY DEPLOYABLE KCFI WITH EBPF

This section focuses on solving the efficiency problem as well as the scalability and coverage issues of kernel extensions in the context of eBPF-based KCFI mechanism, which is prompted by the lack of flexibility in existing KCFI mechanism. While eBPF is a promising mechanism for implementing a flexible and programmable KCFI framework, it is fundementally limited by its existing infrastructure in supporting the needed instrumentations on control flow transfer sites. Specifically, there is no eBPF program type that can satisfy essential requirements of KCFI: (1) complete coverage on all indirect control flow transfers in kernel code, (2) scalable attachment of CFI checks as eBPF programs, (3) low execution overhead of eBPF policy programs. For the program types commonly used for control flow transfer instrumentation, i.e., fprobe-BPF [146] and kprobe-BPF [67], fprobe suffers from coverage issues and inefficiencies from unnecessary execution on direct control flow transfers, while kprobe has to be attached repeatedly to every indirect control transfer instruction in the kernel and suffers from its trap-based design, which incurs significant context switch overhead.

We present eKCFI, an eBPF-based KCFI mechanism that offers policy flexibility, while at the same time achieving efficiency and scalability. eKCFI leverages eBPF to implement programmable KCFI policies—each time an indirect control transfer instruction is reached, the eBPF policy program is invoked to check the validity of the target. This allows not only integration of different KCFI policies from advanced program analysis for stronger CFI protection, but also easy switch of policies at runtime, offering security administrators a useful tool to adjust their defenses without rebooting the kernel. eKCFI also allows policies to configure the temporal (e.g., during a specific time) and spatial (e.g. specific areas in the kernel) aspects of KCFI enforcements, and the action to take upon a check failure.

eKCFI consists of an LLVM transformation pass that inserts instrumentation code at compile time before every indirect control flow transfer checked by the LLVM-KCFI, thus satifies the coverage requirement. The instrumentation code contains a `nop` instruction that can be dynamically patched into a same-sized `call` instruction to the global eKCFI trampoline. After patching, the trampoline will be invoked every time the indirect control flow transfer instruction is reached. The implementation of the eKCFI trampoline satisfies the scalability and efficiency requirements: the eBPF policy program only needs to attach once to the eKCFI trampoline, where it is invoked as a direct function call. Such an implementation that allows dynamically enabling and disabling individual instrumentation is also the key to supporting the temporal and spatial aspects of KCFI enforcements.

We evaluate the efficacy of eKCFI on its flexibility benefits in terms of enhanced security by more precise CFI policies, as well as its performance. Specifically, we use eKCFI to enhance the policy of LLVM-KCFI with a stronger policy from the TyPM analysis [59] and a call-site sensitive policy [75].

Our eKCFI implementation also exhibits a reasonable performance overhead for improved security. For example, supporting the eKCFI-TyPM policy on eKCFI introduces an average of 17% overhead over LLVM-KCFI on LEBench [147].

3.2.1 Motivation

The eBPF extensions on Linux has the potential for implementing a flexible kernel control flow integrity mechanism, due to its programmable and dynamic nature, which allows differnet CFI policies to be implemented and switched at runtime. However, the current eBPF infrastructure is fundementally limited for KCFI, despite existing eBPF program types for kernel instrumentation. For example, fprobe [146] allows an eBPF program to be attached to entry points of multiple kernel functions and executed before the actual function body. However, fprobe-eBPF programs are triggered on every invocation to the target function and thus introduce unnecessary overhead—a direct call does not need to be checked as the target address cannot be corrupted. Moreover, fprobe has incomplete KCFI coverage—a large number of kernel functions (e.g., all functions under the tracing infrastructure) cannot be instrumented with fprobe.

Linux's kernel probe mechanism, kprobe [67], can instrument a given kernel instruction. However, it has a number of limitations. First, as discussed in Section 3.1, kprobe dynamically rewrites an instrumented instruction into a trap instruction. Execution of kprobe-eBPF programs trggers traps and context switches. This trap-based mechanism makes kprobe slow, as context switching to the trap handler incurs significant overhead every time a kprobed instruction is reached. This performance problem can be addressed by recent work on enabling trapless kprobe [46, 74].

Although the performance overhead from traps can be alleviated by recent optimization efforts on Linux kprobe [46] and addressed completely by the universally trapless kernel probe design presented in Section 3.1, the limitation of kprobe persists. Kprobe also incurs scalability issues, because it requires an eBPF program to be attached for *every* instrumented instruction, and thus cannot scale to KCFI that needs to check *all* indirect control flow instructions. Moreover, indirect control flow transfers can exist in the execution of eBPF policy programs, e.g., via helper functions that call deeply into the kernel [77]—the invocation of the kprobe-eBPF program is done by an indirect call. Without protection, the KCFI implementation itself would be vulnerable to control flow hijacking.

Therefore, despite the potential optimizations on Linux kprobe, a new kernel-extension-based KCFI mechanism that achieves the required efficiency, scalability, and coverage.

3.2.2 Threat Model

We strictly follow the threat model of LLVM-KCFI (the *de facto* KCFI mechanism for Linux). We assume that the kernel is benign, but may contain vulnerabilities that allow unprivileged attackers to corrupt kernel control data. We also assume that attackers cannot manipulate permissions of kernel pages, i.e., they cannot modify kernel code or read-only data. We focus on forward-edge control-oriented attacks that overwrite control flow targets of indirect calls and jumps. Following LLVM-KCFI, we do not consider backward-edge control flow attacks in both function returns and context switches, or attacks that modifies interrupt descriptor tables. Unlike forward-edge CFI, whose target precision is still under active research with advancing analysis techniques, backward-edge control flow targets can be captured precisely with a shadow stack, hence its flexibility is less of a concern.

Our use of LLVM for instrumentation and eBPF for enforcement leads us to make assumptions specific to our mechanism. We assume the eBPF program that implements the CFI policy is loaded by a trusted system administrator. We trust the LLVM toolchain and in-kernel BPF-JIT compiler to generate correct code. Similar to LLVM-KCFI [151], we trust the `panic` function to be free of attacks; this function triggers a kernel panic and crashes the kernel when a CFI check fails. We do *not* assume other kernel functions, including all eBPF helper functions, to be free of attacks. Lastly, like LLVM-KCFI, we assume that any kernel modules is available in source code form and are compiled against the running kernel. This assumption ensures loadable kernel modules have the same compiler instrumentations as the core kernel.

3.2.3 Design and Implementation

eKCFI consists of three main components: compiler-based instrumentation (Section 3.2.3), eKCFI kernel runtime enforcement (Section 3.2.3), and eBPF programs that define KCFI policies (Section 3.2.3). eKCFI achieves the desired flexibility in KCFI policies by decoupling the CFG and enforcement policies from the compiled kernel and allowing them to be updated at runtime. Figure 3.6 depicts the workflow of eKCFI. A system administrator can develop the eBPF policy program together with a CFG that contains the valid control flow targets (encoded in an eBPF map). The eBPF policy program and CFG can then be loaded into the kernel and attached to the eKCFI infrastructure. The administrator would then enable eKCFI via its control interface, after which every selected indirect control flow transfer triggers the policy program for validation.

eKCFI address the limitations of the existing Linux eBPF infrastructure. Specifically, it achieves the following three goals:

- **Complete coverage.** eKCFI covers all indirect control flow transfers that are checked by LLVM-KCFI.
- **Scalable attachment.** Different from kprobe, eKCFI attaches eBPF policy programs once, avoiding repetitive attachments for different indirect control flow transfers.
- **Efficient runtime checks.** eKCFI (1) only checks indirect control flow transfers (unlike ftrace) and (2) directly calls eBPF programs without traps (unlike kprobe).

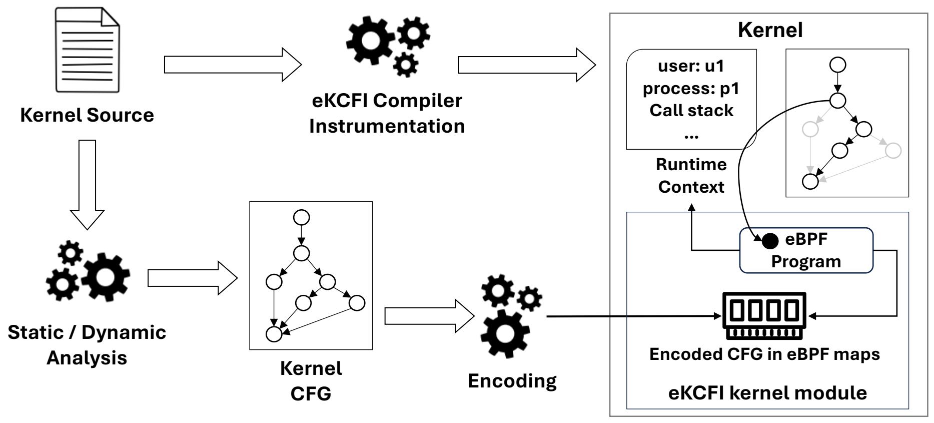


Figure 3.6: Workflow of eKCFI.

Kernel Code Instrumentation

eKCFI includes an LLVM transformation pass to instrument kernel code to achieve complete coverage and efficient runtime checks (Section 3.2.3). It instruments each indirect control flow transfer with code that gathers information about the transfer and efficiently executes the eBPF policy program. Specifically, eKCFI inserts two instructions before each indirect control flow transfer instruction, as exemplified as follows:

```
   48 89 df          mov  %rbx,%rdi
 + 4c 89 d8          mov  %r11,%rax
 + 0f 1f 44 00 08    nopl 0x8(%rax,%rax,1)
   41 ff d3          call *%r11 # indirect call
```

The first instruction moves the control flow target address into the `rax` register. This allows the eKCFI trampoline to always find the target address from `rax` and pass it to the eBPF policy program. The clobbering of `rax` is safe, as `rax` is a caller-saved register. Since eKCFI inserts the two instructions immediately before the indirect control flow transfer, the current function should have already saved the content by spilling it to the stack. The only special case is when `rax` is used to hold the control flow target for the indirect control flow transfer instruction. In this case, the `mov` instruction moves the content of `rax` into itself, which has no effect and does not disrupt the normal execution. The second instruction inserted is a 5-byte `nop` instruction that will be dynamically patched to a same-sized `call` instruction. This `nop` instruction avoids traps and is in principle the same as the one used by ftrace, albeit with a different encoding to distinguish from the ftrace `nops`.

We modify the LLVM KCFI pass in its x86 backend to support the eKCFI instrumentation. Building on top of the LLVM KCFI pass allows us to instrument precisely the same set of indirect control flow transfers checked by LLVM-KCFI, thereby satisfying our coverage requirement. Since the LLVM KCFI pass always loads the control flow target from memory into a register and rewrites the indirect control flow transfer instruction to always use that register, our eKCFI enforcement is free of time-of-check to time-of-use (TOCTOU) issues, as register contents cannot be modified by another context, in contrast to memory content. At the same time, LLVM by default emits 5-byte `nops` with a different encoding (`0f 1f 44 00 08`) from ftrace `nops` (`0f 1f 44 00 00`). This allows us to easily distinguish between eKCFI `nops` and ftrace `nops`.

KCFI Enforcement

The runtime KCFI enforcement is managed by the eKCFI kernel module. The eKCFI module is responsible for dynamically rewriting the compiler-generated `nops`. It also includes the trampoline to execute the eBPF policy program and take post-checking actions. It exposes system calls as control interface for managing these eKCFI policies from user space.

Dynamic text patching. eKCFI leverages the kernel text patching functionality to dynamically rewrite the 5-byte `nop` instruction into a same-sized `call` instruction to the eKCFI trampoline. Rewriting into a `call` allows the eKCFI trampoline to easily retrieve the source and target information about the currently instrumented control flow transfer. One challenge of dynamic code patching in eKCFI is to correctly identify the locations of compiler-generated `nops` for indirect control flow instrumentation. eKCFI employs a trusted user space tool to analyze the compiled kernel image and extract the `nop` locations based on their unique encoding (Section 3.2.3). The list of locations is then sent to the eKCFI module via its control interface. After the initialization, a user can enable instrumentation at selected locations (as defined by the KCFI policy) by passing the addresses to the control interface. The handling of `nop` locations does not break Kernel Address Space Layout Randomization (KASLR): user space can just send static addresses before KASLR to the eKCFI module in the kernel, which can translate the static addresses into real addresses after randomization by adding the KASLR offset. Loadable kernel modules can be handled in the same way, albeit the user space sends the offsets of the `nops` in the module text section to eKCFI. The eKCFI module can then obtain the real address of these `nops` in the kernel by adding the base address at which the text section is mapped to the offsets.

eKCFI trampoline. The eKCFI trampoline serves as a single global landing pad for all dynamically patched `call` instructions at the indirect control flow transfers. This design allows eKCFI to only load and attach a single eBPF program to the trampoline to check all indirect control flows in the kernel.

When the execution enters the eKCFI trampoline, the trampoline first pushes the registers onto the stack. It then loads the target address of the current indirect control flow transfer from `rax` (Section 3.2.3). The source address of the control flow—the address of the indirect control flow transfer instruction—can be retrieved via the return address of the trampoline. In x86, the return address a `call` instruction pushes to the stack always points to the next instruction. For the `call` to the eKCFI trampoline, this instructions is exactly the instrumented instruction. The eKCFI trampoline efficiently invokes the eBPF program via Linux *static calls* [152] with the arguments as the source and target addresses of the indirect control flow.

To prevent recursion (an indirect call in some eBPF helper function used by the eKCFI eBPF program triggering the eKCFI trampoline again), we disable preemption and use a per-CPU flag to record whether an eKCFI eBPF program is currently executing on the CPU. A nested call to eKCFI would find the per-CPU flag to be set. In this case, the trampoline will just return directly without executing the eBPF program.

For enforcement, the eKCFI trampoline implements actions depending on the return value of the eBPF program. The program can return `EKCFI_RET_ALLOW` to signal a passed check and `EKCFI_RET_PANIC` to ask the trampoline to trigger a kernel panic in case of a KCFI failure.

eKCFI control interface. eKCFI exposes a control interface via `ioctl()` to allow administrators to manage eBPF policies and KCFI instrumentations. The `ioctl()` interface implements functionalities that initialize the list of instrumentation locations, attach the eBPF program, and dynamically patch the `nop` instructions. eKCFI only allows a process with root-privilege (the `CAP_SYS_ADMIN` capability) to invoke the `ioctl()` system call, thereby preventing unprivileged and untrusted processes from tempering with the control interface.

eBPF Policy Program

With eKCFI, KCFI policies can be implemented as eBPF programs, together with the CFG encoded from the output of program analysis tools (e.g., the TyPM static analyzer [59] in our evaluation). Once the CFG (e.g., in a hash map) is prepared, the administrator can load the program and attach it through the eKCFI control interface.

Figure 3.7 shows an example eKCFI eBPF program that encodes the CFG as a nested hash map for efficient lookups (L1–7), where the first-level hash map uses the source address of the indirect control flow transfer as keys to index into the second level hash map that stores the valid targets of that indirect control flow transfer. The program takes in the eKCFI specific context struct, which contains both the source and target addresses of the current indirect control flow transfer. The program first looks up the source address from

```c
 1 struct {
 2   __uint(type, BPF_MAP_TYPE_HASH_OF_MAPS);
 3   __uint(max_entries, 16384);
 4   __uint(map_flags, BPF_F_RDONLY_MAPPING);
 5   __type(key, __u64);
 6   __type(value, __u32);
 7 } cfg SEC(".maps");
 8 
 9 SEC("ekcfi") int ekcfi_check(struct ekcfi_ctx *ctx) {
10   // Look up the valid targets from cfg
11   void *tgts = bpf_map_lookup_elem(&cfg, &ctx->src);
12 
13   if (!tgts) // CFG not having this source address
14     return EKCFI_RET_ALLOW;
15 
16   // Look up target from valid targets
17   if (bpf_map_lookup_elem(tgts, &ctx->tgt))
18     return EKCFI_RET_ALLOW;
19 
20   return EKCFI_RET_PANIC;
21 }
```

Figure 3.7: An eKCFI policy program that checks indirect control flow targets based on a CFG encoded in a BPF map.

the CFG to obtain an inner hash map that stores the corresponding list of valid targets (L11). If the current control flow transfer is not captured in the CFG, the control flow is allowed. Otherwise, the program looks up the target address from the inner hash map (L17). A successful lookup in the inner hash map indicates a valid target and the indirect control flow transfer will be allowed. If the lookup fails, the program returns `EKCFI_RET_PANIC` to notify the eKCFI trampoline to trigger a kernel panic to prevent a potential attack, which is the same as the LLVM-KCFI.

The programmable nature of eBPF allows users to implement policies which may encode the CFG differently (e.g., a single level hash map may be more efficient for more restrictive CFGs) or use helper functions to incorporate runtime contexts such as the process information. We demonstrate implementation of addtional KCFI policies in our evaluation (Section 3.2.4).

Protecting eKCFI Infrastructure

As a security enforcement mechanism, the runtime of eKCFI must be protected against corruption and manipulation. We address the protection for the data and code of eKCFI.

Protecting data. As shown in Section 3.2.3, eBPF policy programs can encode CFGs into eBPF maps. Protecting maps is critical because tempering with the map content could lead to valid control flow transfer targets being overwritten to invalid ones, thereby breaking the protection provided by eKCFI. To make the maps tempering-proof, we make the map contents read-only after program attachment. Specifically, we introduce a new eBPF map flag, `BPF_F_RDONLY_MAPPING`. If this flag is set, the kernel allocates the map on a dedicated set of pages. When the policy program is attached to eKCFI, eKCFI iterates over the maps referenced by the program recursively and sets permission of the pages of such maps to read-only.

Protecting code. eKCFI protects its code by avoiding indirect control flow transfers. We address the two notable users of indirect calls/jumps in eKCFI: (1) the invocation of the eBPF policy program, and (2) the helpers called by the program.

First, as eBPF programs are dynamically loaded (therefore not in the kernel symbol table before boot), an naïve invocation is inevitably indirect. eKCFI leverages *static calls* on Linux [152], which transforms indirect calls into direct calls by dynamically patching the call-site when the call target changes. When a policy program is attached, eKCFI updates the static call-site to create a direct call to the program.

Second, eBPF helper functions may call deep into the kernel [77] and therefore may perform indirect control flow transfers. The design of eKCFI only allows helpers that do not use indirect control flows. These include the context retrieval helpers (`bpf_get_current_*`), the map lookup helpers, and the stack trace helper. The context retrieval helpers do not use any indirect control flow transfers. Map lookup helpers are implemented to use indirect calls by default via the `ops->map_lookup_elem` function pointer of the map struct. However, for the maps that are useful in eKCFI, namely the nested hash maps, the specific lookup operation is inlined by the in-kernel eBPF JIT compiler, making it a direct call. The lookup itself is also free of indirect control flow transfers in its implementation.

An exception in eKCFI's design is `bpf_get_stack`, which obtains the current call stack. The helper function is important for call-site sensitive CFI policies [75], which use the call stack to strengthen CFI protection. Internally, the helper uses the kernel stack unwinder (e.g., the ORC unwinder [153]) API. The unwinder is trusted and is used by the kernel `panic` function. eKCFI implements a special version of this helper function to only allow unwinding of kernel stack, avoiding possible indirect control flows from code to obtain user stacks.

3.2.4 Evaluation

We evaluate eKCFI on its flexibility measured by security benefits and its performance. All measurements are performed on a Ubuntu 22.04 LTS server with an 8-core Intel Xeon

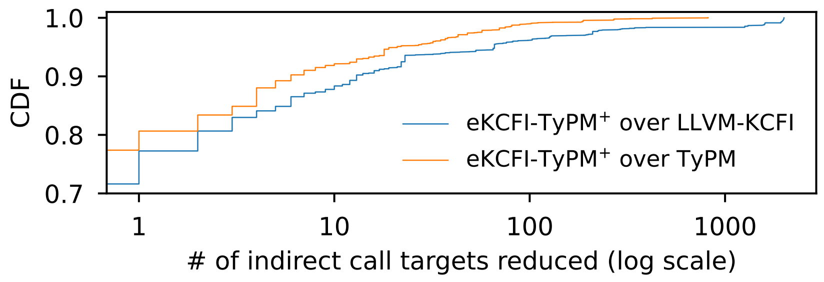


Figure 3.8: CDF of reduced targets at indirect call-sites by eKCFI-TyPM$^+$ over LLVM-KCFI and TyPM. y-axis starts from 70%

E-2174G CPU with 32GB of memory. Linux v6.3.6 is used for all implementation and experimentation; we turn off dynamic CPU-frequency scaling and microarchitectural side-channel mitigations. We use the Ubuntu kernel configuration but with only necessary drivers built-in (via `localyesconfig`).

Flexibility and Security

eKCFI's flexibility is obvious—one can turn on and off CFI enforcement for a deployed kernel at anytime and apply CFI for any selected kernel components or instructions. We show that eKCFI's flexibility can help enhance the security of existing LLVM-KCFI by (1) plugging in more precise CFGs generated by advanced analysis of TyPM [59] and (2) supporting policies that consider call-site [75]. The former shows that eKCFI can leverage independent work that improves CFGs; the latter shows that eKCFI can acquire runtime state.

TyPM CFG. TyPM [59] uses a new type-based analysis that identifies modules that never pass an object of a specific type to a memory-accessing instruction, effectively reducing indirect call targets. TyPM outputs the CFG of indirect calls in the form of call-sites associated with a list of targets; this CFG can be used by the eBPF policy program in Figure 3.7. We correlate the call-sites and targets with its address in the kernel image and encode the CFG into the nested hash map.

We compare the TyPM CFG with the CFG generated by LLVM-KCFI on the v6.3.6 Linux kernel. We consider the same set of call-sites (∼12.7k) in the kernel. More than 28% call-sites in the TyPM CFG have less targets than LLVM-KCFI, while 23% call-sites in LLVM-KCFI has smaller target size. So, we can combine LLVM-KCFI and TyPM to achieve a stronger CFI policy which can be directly supported by eKCFI, referred to as eKCFI-TyPM$^+$. eKCFI-TyPM$^+$ reduces number of targets for 28% call-sites compared to LLVM-KCFI. For these call-sites, eKCFI-TyPM$^+$ reduces 100+ targets for 14% of call-sites and 1000+ targets for 6% of call-sites (Figure 3.8 shows the cumulative distribution).

Call-site sensitivity. We then use eKCFI to implement a call-site sensitive policy [75]. Call-site sensitive CFI takes into account the return address information from the current call stack—a target is only valid if the execution comes from a particular path—and thus further restricting the indirect control flow targets. For example, a *one* call-site sensitive CFI policy takes into the account the return address of the function in which the indirect call instruction (the caller) is located.

We implement a one call-site sensitive CFI policy by modifying the hash map in Figure 3.7 to encode the call stack information—the first level of hash map now uses a stuct that contains both the address of the caller and the return address of the function in which the caller is located as the key. To check the return address, we obtain the return address of the previous stack frame using the `bpf_get_stack` helper function. The program can then use the caller and the return address to index into the hash map and check the validity of the current indirect call. With an analysis that generates the indirect control flow CFG with additional call stack information, such as dynamic analysis or the multi-scope CFG [75], eKCFI can enforce the call-site sensitive policy.

Performance

We use both micro and macro benchmarks to evaluate eKCFI. For micro benchmark, we use LEBench [147] to measure core kernel functions. We use three macro-benchmarks: building the Linux kernel (KBuild), NGINX exercised by ab [127], and Redis exercised by memtier [128]. We use default configurations of the applications and benchmarks; for NGINX and Redis, the clients are on a different machine to avoid interference.

We benchmark the overhead of four KCFI policies: 1) no KCFI, 2) LLVM-KCFI, 3) eKCFI-TyPM$^+$: eKCFI with TyPM on top of LLVM-KCFI that only instruments indirect control flow transfers where TyPM is more restrictive than LLVM-KCFI, and 4) eKCFI-CS: a call-site sensitive policy; the CFG is obtained by dynamically tracing indirect control flows of the kernel executing our benchmark programs.

Figure 3.9 shows the overhead on LEBench under different KCFI policies. LLVM-KCFI generally incurs negligible overhead across LEBench. eKCFI-TyPM$^+$ adds an overhead of 1.17x, with a maximum overhead of 1.95x on `poll`. eKCFI-CS incurs significant overhead, which is 5.45x on average, because `bpf_get_stack` is expensive and needs to be called on every indirect control transfer.

Table 3.5 shows macro benchmark results. LLVM-KCFI only introduces a small performance degradation on all the macro benchmarks in terms of execution time or throughput.

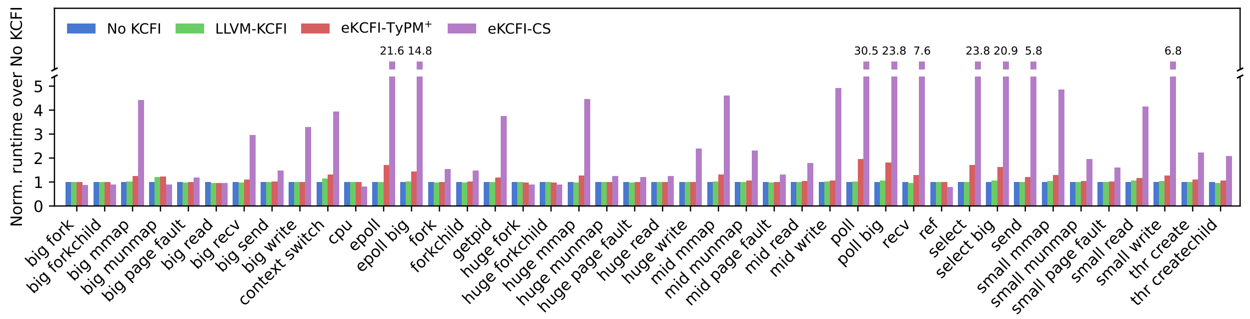


Figure 3.9: Runtime overhead of different KCFI policies on the LEBench microbenchmark. K-best values [147] are reported.

Table 3.5: Application performance with various KCFI policies.

App.	No KCFI	LLVM-KCFI	eKCFI-TyPM$^+$	eKCFI-CS
KBuild	57.2 (second)	+0.49%	+1.14%	+9.08%
NGINX	20.4K (RPS)	−0.06%	−0.13%	−66.7%
Redis	46.5K (RPS)	−1.33%	−2.01%	−82.7%

eKCFI-TyPM$^+$ adds slightly more overhead; overall, it degrades application performance by up to 2.01%. eKCFI-CS incurs major overhead, with a maxmimum 82.7% throughput reduction on Redis. One way to reduce the eKCFI-CS overhead is to selectively enforce it, e.g., only when specific high-risk applications are running. eKCFI offers such flexibility—the eBPF policy program selectively unwinds stacks based on task ID. Our measurements show such process-specific enforcement can reduce impact on non-target applications by 72%.

We also measure the overhead due to compile-time inserted code when eKCFI instrumentation is not enabled. In LEBench, the additional instructions incur negligible (<1%) overhead. We observe a 5.8% bloat of kernel image size on our eKCFI-enabled kernel over a vanilla kernel under the same configuration but without any KCFI instrumentations.

3.2.5 Summary

We discuss the design and implementation of eKCFI, an eBPF-based kernel CFI mechanism that offers policy flexibility. eKCFI eases the adoption of new KCFI policy from new program anlaysis techniques and enables updating KCFI policies at runtime. eKCFI achieves complete CFI coverage, scalable program attachment, and efficient instrumentation. We show the flexiblity from eKCFI can effectively enhance the kernel control flow security, with reasonable overhead. We plan to further optimize eKCFI with hardware-based CFI mechanisms like Intel IBT [154] which only supports coarse-grained CFI policies that ensure indirect control flows land on some targets that are meant to be reached indirectly. eKCFI can provide additional protection at selected locations and meanwhile use hardware-based shadow-stack like Intel CET [155] for backward-edge protection.

CHAPTER 4: PROGRAMMABILITY

Chapter 2 and Chapter 3 presented solutions to the expressiveness and efficiency problem of kernel extensions by advocating and improving the eBPF kernel extensions on Linux. While the eBPF extension mechanism seems promising for a variety of use cases, we have identified significant usability issues with eBPF that make it hard to program: despite the eBPF verifier serving as a guard to prevent unsafe code from being loaded into the kernel, in a lot of cases it incorrectly rejects correct and safe eBPF programs. This problem is also backed by our personal experiences working with eBPF, where frequent verifier rejections create challenges in implementing the programs that fulfill our needs. The root of the problem, which we will discuss later in this chapter, is the mismatch between developers' expectation of program safety provided by a contract with the programming language and the verifier's expectation, which we term the *language-verifier gap*. The language-verifier gap is fundamental to the static verification scheme of eBPF. This inherent limitation on programmability/usability has prompted us to rethink whether eBPF represents a correct design of a safe and usable kernel extension framework, as its static verification apparently trades off programmability. This chapter, therefore, focuses on how kernel extensions can be made usable and easily programmable, while maintaining its important safety properties. We show that with a design on top of language-based safety as well as a lightweight, extralingual runtime, both safety and programmability can be achieved and outweighs the cost of bounding the design to a particular language.

We present Rex, a new kernel extension framework that closes the language-verifier gap and effectively improves the usability of kernel extensions, in terms of programming experience and maintainability. Rex builds up safety guarantees for kernel extensions based on safe language features. With Rex, safety properties are checked by the language compiler within the language contract. Rex drops the need for an extra verification layer and closes the language-verifier gap. We choose Rust as the safe language, as it is already supported by Linux [1] and offers desired language-based safety for practical systems programming [2, 145, 156, 157].

Rex kernel extensions are strictly written in *safe* Rust with selected features (unsafe Rust code is forbidden in Rex extensions). Rex transforms the promises of Rust into safety guarantees for extension programs with the following endeavors. First, to enable Rex extensions to be written entirely in safe Rust in the context of kernel extension, Rex develops a kernel crate and offers a safe kernel interface that wraps the existing eBPF kernel interface (eBPF helper functions and data types) with safe Rust wrappers and bindings. The kernel crate enforces memory safety, extends type safety, and ensures safe interactions with the kernel. Rex further enforces only safe Rust features through its compiler toolchains.

Moreover, Rex employs a lightweight extralingual runtime for safety properties that are hard to guarantee by static analysis. Specifically, Rex supports safe stack unwinding and resource cleanup upon Rust panics at runtime. Rex also checks kernel stack usage and uses watchdog timers to ensure termination with a safe mechanism. The Rex runtime is engineered with minimal overhead to achieve high performance.

We evaluate Rex on both its usability and performance. We show that by closing the language-verifier gap and offering Rust's rich built-in functionality, Rex effectively rules out the usability issues in eBPF. We further evaluate the usability by implementing the BPF Memcached Cache (BMC) [12] (a complex, performance-critical program written in eBPF) using Rex and show that Rex leads to cleaner, simpler extension code. We also conduct extensive macro and micro benchmarks. Rex extensions deliver the same level of performance as eBPF extensions—the enhanced usability does not come with a performance penalty.

This work makes the following main contributions:

- A discussion of the language-verifier gap and its impact on the usability and maintainability of safe kernel extensions;
- Design and implementation of the Rex kernel extension framework, which closes the language-verifier gap by realizing safe kernel extensions upon language-based safety, together with efficient runtime techniques;
- The Rex project is at https://github.com/rex-rs.

4.1 SAFETY OF KERNEL EXTENSIONS

Safety is critical to OS kernel extensions—extension code runs directly in kernel space, and bugs can directly crash a running kernel. The eBPF verifier checks safety properties of extension programs in bytecode before loading them into the kernel to prevent programming errors such as illegal memory access. The verifier also checks the extension's interactions with the kernel via a bounded interface, defined by eBPF *helper functions*, to prevent resource leaks and deadlocks. We summarize the safety properties targeted by eBPF as follows:

- **Memory safety.** Kernel extensions can only access pre-allocated memory via explicit context arguments or kernel interface (helper functions), preventing NULL pointer dereferencing and corruption of kernel data structures.

- **Type safety.** When accessing data in memory, kernel extensions must use the correct types of data, avoiding misinterpretation of the data and memory corruption.
- **Resource safety.** When acquiring kernel resources (e.g., locks, memory objects, etc.) through helper functions, kernel extensions must eventually invoke the appropriate interface to release the resources, preventing memory leaks or deadlocks that can crash or hang the kernel.
- **Runtime safety.** Kernel extensions must terminate, with no infinite loops that can hang the kernel indefinitely.
- **No undefined behavior.** Kernel extensions must never exhibit undefined behavior, such as integer errors (e.g., divide by zero) that cause kernel crashes.
- **Stack safety.** Kernel extensions must not overflow the limited and fixed-size kernel stack, avoiding kernel crashes or kernel memory corruptions.

Note that the above notion of safety in eBPF focuses on preventing programming errors that may crash or hang the kernel. Despite the discussions on whether security is a reasonable target [158, 159], in practice, eBPF and other extension frameworks (e.g., KFlex [7]) no longer pursue unprivileged use cases due to its inherent limitations (see detailed discussion in Section 4.3) [160, 161, 162].

4.2 THE LANGUAGE-VERIFIER GAP

A fundamental problem of eBPF's safety verification mechanism is the *language-verifier gap* (illustrated in Figure 4.1). Developers mainly implement and maintain eBPF extensions in high-level languages (e.g., C and Rust) and compile them to eBPF bytecode; they agree to a contract with the high-level language (code has property p), which is enforced by the compiler. When a program fails to compile, developers receive feedback about how they violate the language contract. However, in eBPF, the compiled extension code will be further checked by the verifier. If correctly compiling code fails to pass the verifier (e.g., the code lacks property q, which is not part of the contract), it is difficult for developers to understand why the extension program fails despite obeying the contract.

The language-verifier gap is further exacerbated when the verifier incorrectly rejects *safe* extension programs due to (1) scalability limitations of the symbolic execution used by the verifier, (2) conflicting analyses between the compiler and the verifier, and (3) the verifier's implementation defects. Today, the language-verifier gap forces developers to understand

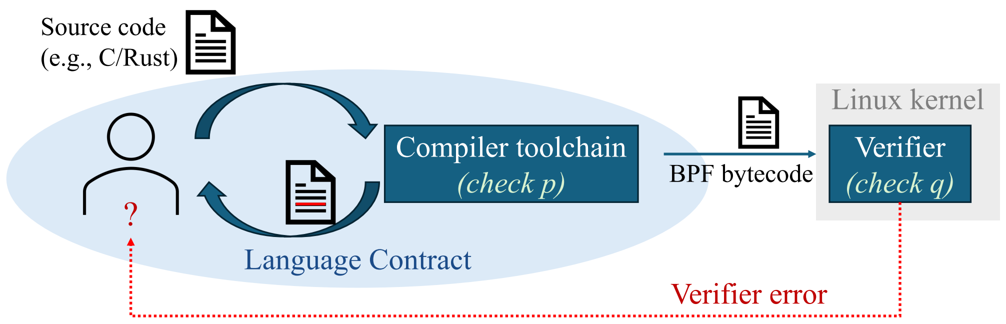


Figure 4.1: The language-verifier gap

the verifier's internal implementation and its limitations and defects, and to revise extension code in ways that can pass the verifier at the bytecode level. Many such revisions are workarounds solely to please the verifier. Fundamentally, the language-verifier gap breaks the language abstractions and artificially forces extension programs in a high-level language to tightly couple with the low-level verifier implementation.

4.2.1 Verifier Workarounds

To understand the impact of the language-verifier gap, we analyzed commits related to revising eBPF programs to resolve verifier issues in popular eBPF projects, including Cilium [163], Aya [164], and Katran [165]. The commits were collected by searching through the commit logs of each project using keywords and manually inspected. In total, we collected 72 commits related to verifier issues. We also included two issues raised by BMC [12] and Electrode [16] in the discussion.

In all 72 commits, we confirmed that the original eBPF programs were safe but were rejected by the verifier due to defective or overly conservative safety checks. When the verifier rejects a safe program, the developer must find a *workaround*. Table 4.1 summarizes the workaround patterns.

Refactoring Extension Programs into Small Ones

The most common pattern (27 out of 72) is refactoring a large eBPF program, which the verifier rejects due to exceeding the verifier's internal limits into smaller ones. Since symbolic execution is hard to scale, the eBPF verifier imposes a series of limits on the complexity of extension programs (e.g., the number of bytecode instructions and branches [166]) to ensure verification to complete at load time. The eBPF extension will be rejected if it exceeds any

Table 4.1: Patterns of common verifier workarounds

Category	**Count**
Refactoring extension programs into small ones	27
Hinting LLVM to generate verifier-friendly code	22
Changing code to assist verification	15
Dealing with verifier bugs	9
Reinventing the wheels	1

of these limits. Such rejections have no implication on the safety of the extension; rather, they are artifacts of scalability limitations of static verification.

We observe two standard practices of refactoring eBPF programs to work around verifier limits: (1) splitting eBPF programs into smaller ones and (2) rewriting eBPF programs with reduced complexity the verifier can handle.

We use BMC [12] as an example to explain these practices. BMC uses eBPF to implement in-kernel caches to accelerate Memcached. Conceptually, only two extension programs are needed (at ingress and egress, respectively). However, to satisfy the verifier limit, BMC developers had to split BMC code into *seven* eBPF programs connected via tail calls.[1] Such splitting creates an unnecessary burden on the implementation and maintenance of BMC; it also creates performance issues when states need to pass across tail calls (using maps).

Despite the smaller size of each program after splitting, BMC programs that iterate over the packet payload in a loop cannot easily pass the verifier. While the programs correctly check for the bounds of the payload, the programs result in an excessive number of jump instructions and exceed the verifier's complexity limit. As a workaround, developers must bind the size of the data BMC can handle further to pass the verifier. Section 4.6.1 revisits this example in more depth.

Hinting LLVM to Generate Verifier-friendly Code

Another common pattern is to change source code in ways that nudge the compiler (LLVM) to generate verifier-friendly bytecode. In several cases, LLVM generated eBPF bytecode that fails the verifier due to complex, often undocumented expectations of the verifier. Figure 4.2 shows a case from Cilium [167] that accesses a pointer field (`ctx->data`) in a socket buffer, defined as a 32-bit integer in the kernel `uapi` interface. LLVM generates a 32-bit load on `data` and assigns its value to another 32-bit register. While `data` is defined as 32-bit, under the hood it represents a pointer to the start of the packet payload. The 32-bit assignment

[1]Since BMC, the limit has increased, but the fundamental gap remains.

```
; return (void *)(unsigned long)ctx->data;
2: (61) r9 = *(u32 *)(r7 +76)
; R7_w=ctx(id=0,off=0,imm=0)
; R9_w=pkt(id=0,off=0,r=0,imm=0)
3: (bc) w6 = w9
; R6_w=inv(id=0,umax_value=4294967295,
;           var_off=(0x0; 0xffffffff))
; R9_w=pkt(id=0,off=0,r=0,imm=0)
...
7: r2 = *(u8 *)(r6 +22)
; R6 invalid mem access 'inv'
```

Figure 4.2: Verifier log showing an invalid memory access, which is hard to diagnose and does not directly map to the source code in C

```
static __always_inline void *
ctx_data (const struct __sk_buff *ctx) {
  void *ptr;
  asm volatile (
    "%0 = *(u32 *)(%1 + %2)"
    : "=r"(ptr)
    : "r"(ctx),
      "i"(offsetof(struct __sk_buff, data))
  );
  return ptr;
}
```

Figure 4.3: Inline assembly code created to work around the verification failure by preventing the compiler optimization

made the verifier interpret the pointer as a scalar and incorrectly reject the program when it tries to access memory through the scalar. As a workaround, developers encapsulated access to `data` in inline assembly (Figure 4.3) to prevent LLVM from generating 32-bit move as an optimization (LLVM does not optimize inlined assembly). The verifier then treats the register as a pointer rather than a scalar.

In another case [168], developers were forced to use `volatile` when loading from a 32-bit integer pointer and only using its upper 16 bits. Without `volatile`, LLVM optimized the code to only load the upper 16-bit from the pointer, which the verifier perceives as a size mismatch violation.

In fact, many eBPF programs today can only pass the verifier if compiled with `-O2` optimization—the verifier has a hardwired view of eBPF extension bytecode, which the compiler cannot generate with other levels, including `-O0`.

Changing Code to Assist Verification

In this pattern, developers had to assist the verifier manually. A common pattern is refactoring the code into new functions when the verifier loses track of values in eBPF programs. It is often unclear what code needs to be refactored to pass the verifier, which significantly burdens developers. Figure 4.4a shows a code example from Cilium [169], which originally used a `goto` statement to combine the code path of `policy` and `l4policy` to avoid duplicated code. However, the combined code, which assigns `l4policy` to `policy`, later causes the verifier to incorrectly believe that `policy`, which is a pointer variable, is instead a scalar and reject the program. As a workaround, developers had to refactor the policy check code into an inlined function to separate the code path to pass the verifier.

```
  if (likely(l4policy && !l4policy->wildcard_dport)) {
    *match_type = POLICY_MATCH_L4_ONLY;
-   policy = l4policy;
-   goto policy_check_entry;
+   return account_and_check(ctx, l4policy, ...);
  }

  if (likely(policy && !policy->wildcard_protocol)) {
    *match_type = POLICY_MATCH_L3_PROTO;
-   goto policy_check_entry;
+   return account_and_check(ctx, policy, ...);
  }
```

(a) Simplifying control flow

```
  #ifndef ENABLE_SKIP_FIB
  ...
  if (likely(ret == BPF_FIB_LKUP_RET_NO_NEIGH)) {
    nh_params.nh_family = fib_params->l.family;
    ...
  } else {
    return DROP_NO_FIB;
  }
  ...
  skip_oif:
  #else
  *oif = DIRECT_ROUTING_DEV_IFINDEX;
- nh_params.nh_family = fib_params->l.family;
  #endif /* ENABLE_SKIP_FIB */
  ...
- dmac = nh_params.nh_family == AF_INET ? ...;
+ dmac = fib_params->l.family == AF_INET ? ...;
```

(b) Simplifying data flow

Figure 4.4: Examples that developers had to work around the language-verifier gap by refactoring already safe extensions

Developers also have to teach the verifier by providing additional information. Figure 4.4b shows an example in Cilium [170] where the verifier lost track of `nh_params.nh_family`, a scalar spilled onto the stack and mistakenly treated it as a pointer when loading it back, leading to an invalid size error on the load. As a workaround, developers passed `fib_params->l.family` directly instead of going through `nh_params.nh_family` to let the verifier know the scalar value.

Dealing with Verifier Bugs

The language-verifier gap is further exacerbated by verifier bugs [77, 171, 172, 173], as developers need to acquire knowledge of the verifier's expectations and deficiencies. Moreover, different kernel versions can have different verifier bugs. Dealing with verifier bugs and maintaining compatibility across kernel versions is non-trivial. In a Cilium case [174], The verifier rejected a correct program with valid access to the context pointer due to the verifier's incorrect handling of constant pointer offsets. The verifier bug was known, but the fix was not present in all kernel versions. Cilium developers had to tweak their program to avoid the bug-triggering yet correct context pointer access so the code could verify on all kernel versions.

Reinventing the Wheels

Developers may need to reimplement existing functions to pass the verifier. In Aya, the default definition of the `memset` and `memcpy` intrinsics provided by the language toolchain failed to pass the verifier [175]. Aya eventually implemented its own version for both intrinsics, using a simple loop to iterate over the data to avoid ever tripping the verifier. This case reflects a key challenge of using eBPF for large, complex extension programs, as developers may need to re-implement many standard, nontrivial library functions.

4.2.2 Implications

Our analysis shows that the language-verifier gap causes severe usability issues in developing and maintaining eBPF kernel extensions. eBPF developers have to implement arcane fixes and change their mental model to meet the verifier's constraints. If an eBPF extension fails to verify, the verifier log rarely pinpoints the root causes and cannot help trace back to the source code.

Unfortunately, recent efforts to improve the eBPF verifier (e.g., via testing and verification [171, 176]) cannot fundamentally close the language-verifier gap because (1) they do not address scalability issues of symbolic execution, so extension programs have to ill-fit the verifier's internal limits, and (2) it is unlikely that the language compiler (e.g., LLVM) and the eBPF verifier are always in synchronization, given their independent developments. Recent efforts to improve extension expressiveness via techniques like software fault isolation (e.g., KFlex [7]) largely inherit the eBPF verifier and, therefore, do not address the language-verifier gap.

4.3 KEY IDEA AND SAFETY MODEL

The key idea of Rex is to realize *safe* kernel extensions without a separate layer of static verification. Our insight is that the desired safety properties of kernel extensions can be built on the foundation of language-based properties of a safe programming language like Rust, together with extralingual runtime checks. In this way, the in-kernel verifier can be dropped, and the language-verifier gap can be closed. Rex extensions are strictly written in a *safe* subset of Rust. We choose Rust as the safe language for kernel extensions (instead of other languages like Modula-3 [4] and Sing# [177]) because Rust is already supported by Linux [178] and offers desired language features for practical kernel code [156, 157, 179]. Rex enforces the same set of safety properties eBPF enforces (Section 4.1). Hence, Rex extensions fundamentally differ from unsafe kernel modules.

Safety Model

Rex follows eBPF's non-adversarial safety model—the safety properties focus on preventing programming errors from crashing/hanging the kernel instead of malicious attacks. Like eBPF, Rex extensions are installed from a trusted context with root privileges on the system. Rex extensions can only be written in safe Rust with selected features and language-based safety is enforced out by a trusted Rust compiler (Section 4.4.1). Unlike Rust kernel modules that can use unsafe Rust, the language-based safety of Rex extensions is strictly enforced. Other safety properties that are not covered by language-based safety (e.g., termination) are checked and enforced by the lightweight Rex runtime.

While historically eBPF supported unprivileged mode [160] and there are research efforts in supporting unprivileged use cases for kernel extensions [71, 158, 159], in practice, eBPF and other frameworks (e.g., KFlex [7]) no longer pursue it [161, 162]. The reasons come from inherent limitations of securing eBPF or kernel extensions in general.

First, it is hard for the eBPF verifier to prevent transient execution attacks like Spectre attacks completely, without major performance and compatibility overheads (see [161]). Specifically, new Spectre variants are being discovered; though many of them are bugs in hardware, they cannot be easily detected and fixed by static analysis [180]. Sandboxing techinques cannot completely prevent Spectre attacks either, e.g., SafeBPF [159] only prevents memory vulnerabilities, while BeeBox [158] only focuses on two Spectre variants and requires manual instrumentation of helper functions. For these reasons, the Linux kernel and major distributions also have moved away from unprivileged eBPF [162, 181, 182].

Second, eBPF chose not to be a sandbox environment (like WebAssembly or JavaScript) that does not know what code will be run [161]. Instead, the development of eBPF assumes that "*the intent of a BPF program is known* [161]."

Lastly, the constantly reported verifier vulnerabilities [77, 183, 184] indicate that a bug-free verifier is hard in practice.

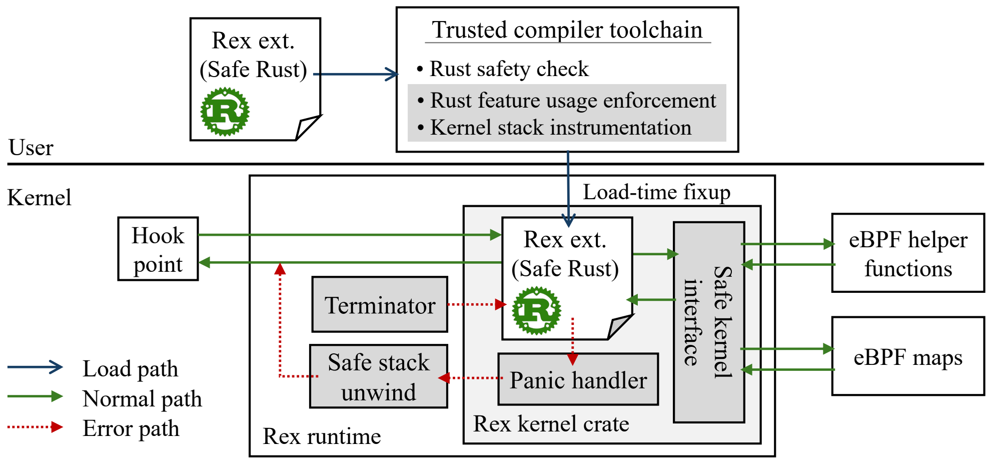


Figure 4.5: Overview of the Rex kernel extension framework. The gray boxes are Rex components.

Trusted Computing Base (TCB)

With Rex's safety model, the TCB consists of the Rust toolchain, the Rex kernel crate, and the Rex runtime. Rex has to trust the Rust toolchain for its correctness to deliver language-based safety. We believe the need to trust the Rust toolchain is acceptable and does not come with high risks with our safety model. Recent work on safe OS kernels [2, 156, 157] makes the same decision to establish language-based safety by trusting the Rust toolchains. The active effort on extensive fuzzing and formal verification of the Rust compiler [185, 186, 187, 188, 189, 190] may further reduce the risk. Certainly, we acknowledge that the existing Rust compiler, such as rustc [191], is larger than the eBPF verifier.

4.4 REX DESIGN

The key challenge of the Rex design is to provide safety guarantees of kernel extensions (listed in Section 4.1) on top of Rust's safe language features (adopting a safe language alone is insufficient, as in Rust kernel modules).

Figure 4.5 gives an overview of the Rex framework. To realize language-based safety, Rex enforces kernel extensions to be strictly written in *safe* Rust with selected features. The Rex compiler toolchain rejects any Rex program that uses unsafe language features. Although this safe subset of Rust already provides inherent language-based safety within Rex extensions, eliminating undefined behaviors, safety of extensions is only achieved with the presence of safe kernel interactions provided by the *Rex kernel crate*. The kernel crate is trusted and bridges Rex extensions with unsafe kernel code. Rex builds on top of the eBPF helper functions interface to provide a safe kernel interface for Rex extensions to interact with the kernel using safe Rust wrappers and bindings. The safe interface encapsulates the interaction across the foreign function interface in the kernel crate. We reuse the eBPF helper interface, because it is designed for kernel extensions with a clearly defined programming contract and separates extensions from kernel's internal housekeeping (e.g., RCU [192]).

Rex employs a lightweight extralingual in-kernel runtime that checks safety properties that are hard for static analysis. The runtime enforces program termination, kernel stack safety, and safe handling of runtime exceptions (e.g., Rust panics).

4.4.1 Safe Rust in Rex

Rex only allows language features that are safe in the context of kernel extensions. First, Rex excludes any *unsafe* Rust code as it misses important safety checks from the Rust compiler and can voilate various safety properties (Section 4.1). Second, Rex also forbids Rust features that interfere with Rust's automatic management of object lifetimes, which include `core::mem::{forget,ManuallyDrop}` and the `forget` intrinsic. These features are considered safe in Rust but violate resource safety of kernel extensions by facilitating resource leakage. Third, language features that cannot be supported in the kernel extension context are excluded by Rex. This group contains the `std` [193] library and dynamic allocation support (not available in `no_std` build [194]), the floating point and SIMD support (generally cannot be used in kernel space), and the `abort` intrinsic (triggers an invalid instruction). Note that dynamic allocation may be supported by hooking `alloc` [195] crate to the kernel allocator. We plan to explore the use of dynamic allocation in kernel extensions in future works (Section 4.7).

To enforce the restrictions, Rex configures the Rust compiler and linter to reject the use of prohibited features. Specifically, we set compiler flags [196] to forbid unsafe code and unstable Rust features which includes SIMD and intrinsics. For individual langauge items such as `core::mem::forget`, we configure the Rust linter [197] to detect and deny their usage. We further remove floating point support by setting the target features [198] in Rex compilation. The `std` library and dynamic allocation are already unavailable in `no_std` environment used by Rex and therefore warrant no further enforcement.

4.4.2 Memory safety

Rex enforces extensions to access kernel memory safely. There are two common memory access patterns, depending on the ownership of the memory region: (1) memory owned by the extension (e.g., a stack buffer) is sent to the kernel through helper functions, and (2) memory owned by the kernel (e.g., a kernel struct) is accessed by the extension.

Memory Owned by Extensions

A Rex extension can allocate memory on the stack and send it to the kernel (e.g., asking the kernel to fill a stack buffer with data) via existing eBPF helper functions. Rex ensures no unsafe memory access and thus prevents stack buffer overflow and kernel crash (e.g., corruption of the return address on the stack).

Unlike eBPF that checks a memory region with its size of every invocation of helper functions, in Rex, the strict type system of Rust already prevents unsafe access. Rex leverages the generic programming feature of Rust [199] to ensure that the size sent through the helper function interface is always valid. For helper functions that take in pointer and size as inputs, the Rex kernel crate creates an adaptor interface that parametrizes the pointer type as a generic type parameter. The interface queries the size of the generic type from the compiler and invokes the kernel interface with this size as an argument. Since Rust uses *monomorphization* [200], the concrete type and its size are resolved at compile time, adding no runtime overhead. In this way, the size is guaranteed to match the type statically and the kernel will never make an out-of-bound access. This works for both scalar types and array types. We use Rust's *const generics* to allow a constant to be used as a generic parameter [199] to encode array lengths.

Memory Owned by the Kernel

The kernel can provide extensions with a pointer to kernel memory (e.g., map value pointers and packet pointers). The extension must not have out-of-bound memory access. In eBPF, the verifier checks uses of kernel pointers with a static size, e.g. map value pointers (maps store the size of values); for pointers without a static size like packet data pointers, the verifier requires extensions to explicitly check memory boundaries.

In Rex, pointers with static sizes are handled through the Rust type system. The kernel map interface of Rex encodes the key and value types through generics and returns such pointers to extension programs as safe Rust references. To manage pointers referring to dynamically sized memory regions, the Rex kernel crate abstracts such pointers into a Rust *slice* with dynamic size. Rust slice provides runtime bounds checks (Section 4.4.5), which allows the check to happen without explicit handling by the extension.

Rust slices are in principle similar to `dynptr` in eBPF [201], but provide more flexibility.

eBPF `dynptr`s are pointers to dynamically sized data regions with metadata (size, type, etc); however, access to the `dynptr`'s referred memory must be of a static size. Rust slices allow dynamically sized access to the underlying memory, benefiting from its runtime bounds checks. Moreover, the `bpf_dynptr_{read,write}` helpers do not implement a zero-copy interface available in Rust slices. While `bpf_dynptr_{data,slice}` helpers allow extensions to obtain data slices without copying, they again require explicit checks of the bound of the slice. As a tradeoff, eBPF `dynptr`s avoids runtime overheads of dynamic bounds checks, which we find negligible in our evaluation (Section 4.6.2).

4.4.3 Extended type safety

Rex extends Rust's type safety to allow extension programs to safely convert a byte stream into typed data. The pattern is notably found in networking use cases, where extensions need to extract the protocol header from a byte buffer in the packet as a struct. Safety of such transformations is beyond Rust's native type safety because they inevitably involve unsafe type casting. eBPF allows pointer casting; the verifier ensures: (1) the program does not make a pointer from a scalar value, and (2) the new type fits the memory boundary.

Rex also enforces the above two properties so that the reinterpreting cast (dubbed "transmute" in Rust) is safe. Rex extends Rust's type safety to cover such casts. To satisfy (1), Rex ensures the target Rust type of casting does not contain raw pointers or safe references in any transitively reachable fields. Rex introduces a Rust *auto trait* [202], `rex::NoRef`. The Rust compiler automatically implements an auto trait on a type unless the type is explicitly opted out via a *negative implementation* [203] or the type contains a field on whose type the trait is not implemented. We negatively implement `rex::NoRef` on the raw pointer and safe reference types of Rust, which ensures any type transitively containing a pointer or a reference will not have an implementation of the trait generated by the Rust compiler. Note that polymorphic types without statically known fields are not a problem because they take the form of *trait objects* [204] in Rust and can only exist behind pointers that already do not have an implementation of `rex::NoRef`. By requiring the target type of casting to implement `rex::NoRef` via trait bounds [205], (1) is effectively satisfied. To satisfy (2), Rex performs explicit bound checking in the transmute interface to ensure the target type always fits.

4.4.4 Safe resource management

Rex extensions are ensured to acquire and release resources properly to avoid leaks of kernel resources (e.g., refcounts and spinlocks). Different from eBPF where the verifier checks all possible code paths to ensure the release of acquired resources, Rex uses Rust's Resource-Acquisition-Is-Initialization (RAII) pattern [206]—for every kernel resource a Rex extension may acquire, the Rex kernel crate defines an RAII wrapper type that ties the resource to the lifetime of the wrapper object.

For example, when the program obtains a spinlock from the kernel, the Rex kernel crate constructs and returns a *lock guard*. The lock guard implements the RAII semantics through the `Drop` trait [207] in Rust, which defines the operation to perform when the object is destroyed. In the case of lock guard, its `drop` handler releases the lock. Rex uses compiler-inserted `drop` calls at the end of object lifetime during normal execution, and implements its own resource cleanup mechanism (Section 4.4.5) for exception handling. The use of RAII automatically manages kernel resources to ensure safe acquisition and release. Extension programs do not need to explicitly release the lock or drop the lock guard.

4.4.5 Safe exception handling

While certain Rust safety properties are enforced statically by the compiler, the others are checked at runtime and their violations trigger exceptions (i.e., Rust panics). To handle exceptions in userspace, Rust uses the Itanium exception handling ABI [208] to unwind the stack. A Rust panic transfers the control flow to the stack unwinding library (e.g., llvm-libunwind), which backtracks the call stack and executes cleanup code and catch clauses for each call frame. Unfortunately, this ABI is unsuitable for kernel extensions:

- Unlike in userspace where failures during stack unwinding crash the program,[2] stack unwinding in kernel extensions cannot fail—kernel extensions must not crash the kernel and must not leak kernel resources.
- Unwinding generally executes destructors for all existing objects on the stack, but executing untrusted, user-defined destructors (via the `Drop` trait [207] in Rust) is unsafe.

Rex implements its own exception handling framework with two main components: (1) graceful exit upon exceptions, which resets the context, and (2) resource cleanup to ensure release of kernel resources (e.g., reference counts and locks).

Graceful Exit

To ensure a graceful exit from an exception, Rex implements a small runtime (Figure 4.6)

[2]Theseus [156] implements stack unwinding in the kernel. But, it assumes that unwinding never fails; faults in unwinding result in kernel failures.

in the kernel, which consists of a program dispatcher, a panic handler, and a landingpad. The dispatcher takes the duty of executing the extension program (like the eBPF dispatcher). It saves the stack pointer of the current context into per-CPU memory, switches to the dedicated program stack (Section 4.4.6), sets the termination state (Section 4.4.7), and then calls into the program. If the program exits normally, it returns to the dispatcher, which switches the stack back and clears the termination state. Under exceptional cases where a Rust panic is triggered, the panic handler releases kernel resources currently allocated by the extension, and transfer control to the in-kernel landingpad to print debug information to the kernel ring buffer and return a default error code to the kernel. Then, the landingpad redirects control flow to a pre-defined label in the middle of the dispatcher, where it restores the old value of the stack pointer from the per-CPU storage. This effectively unwinds the stack and resets the context as if the extension returned successfully.

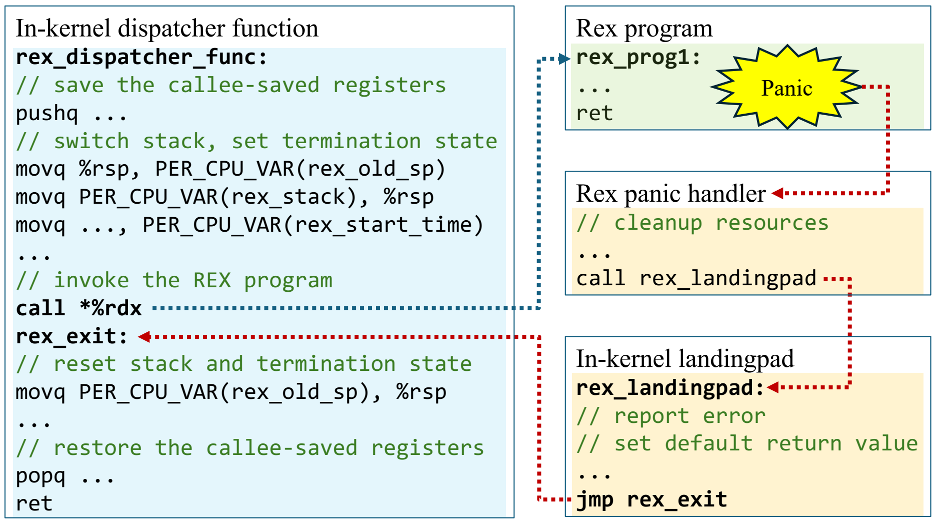
Resource Cleanup

Correct handling of Rust panics requires cleaning up resources acquired by the extension. However, static approaches that rely on the verifier to pre-compute resources to be released during verification (e.g., object table in [7]) do not apply to Rex due to the language-verifier gap.

Our insight is that extensions can only obtain resources by explicitly invoking helper functions. So, Rex records the allocated kernel resources during execution in a per-CPU buffer, which is in principle like the global heap registry in [157]. Upon a panic, the panic handler takes the responsibility to correctly release kernel resources, which involves traversing the buffer and dropping recorded resources.

We implement the cleanup code as part of the panic handler in the Rex kernel crate, as it is responsible for coordinating helper function calls that obtain kernel resources. Implementing the cleanup mechanism in the kernel crate ensures safety: as the code is called upon panic, it must not trigger deadlocks or yet another Rust panic to fail panic handling. The careful design of the Rex kernel crate frees the cleanup code and `drop` handlers of locks and panic-triggering code. Kernel functions invoked by such code may still hold locks internally, but they are self-contained and do not propagate to Rex (deadlocks in kernel functions is out of the scope of Rex). Rex does not execute user-supplied `drop` handlers upon panic, as they are not guaranteed to be safe under panic handling context.

Rex implements a crash-stop failure model—a panicked extension is removed from the kernel. Any used maps and other Rex extensions sharing the maps will also be removed recursively. This prevents extensions sharing the maps from running in a potentially incon-



```
rex_dispatcher_func:
// save the callee-saved registers
pushq ...
// switch stack, set termination state
movq %rsp, PER_CPU_VAR(rex_old_sp)
movq PER_CPU_VAR(rex_stack), %rsp
movq ..., PER_CPU_VAR(rex_start_time)
...
// invoke the REX program
call *%rdx
rex_exit:
// reset stack and termination state
movq PER_CPU_VAR(rex_old_sp), %rsp
...
// restore the callee-saved registers
popq ...
ret
```

```
rex_prog1:
...
ret
```

```
// cleanup resources
...
call rex_landingpad
```

```
rex_landingpad:
// report error
// set default return value
...
jmp rex_exit
```

Figure 4.6: Exception handling control flow in Rex

sistent state—exception handling in Rex already ensures the kernel is left in a good state.

4.4.6 Kernel stack safety

Kernel extensions should never overflow the kernel stack. Unlike userspace stacks which grow on demand with a large maximum size, the stack in kernel space has a fixed size (4 pages on x86-64). The eBPF verifier checks stack safety by calculating stack size via symbolic execution. However, it is reported that stack safety is broken in eBPF due to the difficulties of statically analyzing indirect tail calls [183] and uncontrolled program nestings [209].[3]

Our insight is that stack safety can be enforced at compile time to avoid runtime overhead if the extension program has no indirect or recursive calls, as the stack usage can be statically computed. Otherwise, it is easy to check stack safety at runtime. Rex, therefore, takes a hybrid approach and selects between static and dynamic checks based on the situation.

Static Check

The static check is done by a Rex-specific compiler pass (Section 4.5). If the extension has no indirect or recursive calls, its total stack usage can be calculated by traversing its global static callgraph and sum up the size of each call frame. We turn on the "fat" link time optimization and use a single Rust codegen unit [198] for Rex programs to ensure the compiler always has a global view across all translation units.

[3]Rex currently does not support program nesting (same as eBPF)

Runtime Check

For extensions with indirect or recursive calls, it is hard to calculate the stack usage from the callgraph due to the presence of unknown edges (indirect calls) and cycles (recursive calls). Under these cases, Rex performs runtime checks. The Rex compiler pass first ensures each function in the program takes less than one page (4K) of stack. This is more relaxed than the frame size warning threshold (2K) in Linux and ensures enough stack to handle Rust panics. Before each function call in the extension, the compiler inserts a call to the `rex_check_stack` function from the kernel crate to check the current stack usage: if the stack usage exceeds the threshold, it will trigger a Rust panic and terminate the program safely (Section 4.4.5).

To manage stack usage of Rex extensions effectively, Rex implements a dedicated kernel stack for each extension. The dedicated stacks are allocated per-CPU and virtually mapped at kernel boot time with a size of eight pages. Before executing a Rex extension, the dispatcher (Figure 4.6) saves the stack pointer of the current context, and then sets the stack and the frame pointer (already saved with other callee-saved registers) to the top of the dedicated stack. When the extension exits, the original stack and frame pointers are restored.

Rex sets the stack usage threshold to be four pages for extension code; it reserves the next four pages with following considerations: (1) helper functions are not visible at compile time but they also account for stack usage during execution; we use four pages as the de facto stack size used by the kernel itself, and (2) since stack usage of each function is limited to one page of stack, in the worse case, the remaining stack space is at least three pages when `rex_check_stack` triggers a Rust panic. Since the panic handler is implemented in the kernel crate and does not change with programs, this worse-case guarantee empirically leaves enough space for panic handling and stack unwinding. Rex's dynamic approach achieves stronger stack safety than that of eBPF.

4.4.7 Termination

Termination is an important property of kernel extensions. In eBPF, an extension with a back edge or exceeds the instruction limit will be rejected, regardless whether it eventually terminates. KFlex [7] lifts the back edge restriction by inserting cancellation points in eBPF bytecode on all back edges during verification, which triggers termination at runtime. However, back edge analysis is non-trivial outside eBPF bytecode and is unreliable for general Rust programs.

Rex employs a runtime that interrupts and terminates extensions that run for too long. Rex limits the run time of extensions by leveraging kernel timers as watchdogs. Rex builds the runtime on the high resolution timer (`hrtimer`) subsystem in Linux [210]. Since `hrtimer` callbacks execute in hardware timer interrupts, they are capable of interrupting the contexts in which most extensions execute (soft interrupts and task context [211]). Since hardware timer interrupts are periodically raised by the processor, regardless whether an `hrtimer` is present, executing timer callbacks in this existing hardware timer interrupts adds no extra interrupt or context switch, keeping the watchdog overhead minimal.

Rex sets one timer for each CPU to avoid inter-core communication, in contrast to using a single, global timer to handle programs from all CPUs. Each timer only needs to monitor extensions running on the core. Rex arms the timers at kernel boot time, which are triggered periodically with a constant timeout, and re-armed each time after firing.[4]

Rex implements its watchdog logic in the timer handlers. When a timer fires, its handler suspends any soft interrupt or task context, and saves its registers. The handler then checks the current CPU on whether the termination timeout of the Rex extension in the stopped context has been reached. This is done by comparing the extension start time (stored as a per-CPU state as shown in Figure 4.6) with the current time. If the extension exceeds the threshold, the timer handler overwrites the saved instruction pointer register to the panic handler (Section 4.4.5). After returning from the timer interrupt, the extension executes its panic handler, which cleans up kernel resources and gracefully exits. Rex sets both the timer period and runtime threshold to the default RCU CPU stall timeout (Rex extensions run in RCU locks as they use eBPF hook points).

Rex defers termination when the extension is running kernel helper functions to avoid disrupting the kernel's internal resource bookkeeping; it also does not terminate an extension if it is in the panic handler. Rex uses a per-CPU tristate flag to track the state of an extension: (1) executing extension code, (2) executing kernel helpers or panic handlers, and (3) termination requested. A helper call changes the state from 1 to 2. When executing the timer handler, if the flag is at state 2, the termination handler modifies it to state 3 without changing the instruction pointer. When a helper returns, if the flag is at state 3, the panic handler is called to gracefully exit.

A corner case of this design is deadlock. Since spinlock acquisition in Rex is implemented by a kernel helper function, a deadlocked program will never return from the helper, and therefore will never be terminated properly. Rex follows eBPF's solution toward deadlocks, where a program can only take one lock at a time. This is achieved by using a per-CPU

[4]Disarming the timer when no extension is running saves CPU cycles, but incurs high overhead due to timer setup on execution hot path, especially for frequently triggered extensions (e.g., XDP extensions [6])

variable to track whether the program currently holds a lock—a program trying to acquire a second lock will trigger a Rust panic. We note that if the ability of holding multiple locks at the same time is desired, the kernel spinlock logic can be modified to check the termination state of Rex programs during spinning and terminate a deadlocked program accordingly.

Limitation

Rex uses hard interrupts, and thus cannot interrupt extensions that are already executing in hard or non-maskable interrupts [211] (e.g., hardware perf-event programs). Such extensions are not targeted by Rex, as they are supposed to be small, simple, and less likely to encounter the language-verifier gap. Note that Rex extensions and eBPF extensions are not mutually exclusive and can co-exist.

Moreover, the termination of a timed-out Rex extension can be delayed if the extension is already interrupted by another event when the timer triggers (the registers will not be available to the timer handler). Rex needs to wait for a triggering of the timer that directly interrupts the extension.

4.5 IMPLEMENTATION

We implement Rex on Linux v6.11. Rex currently supports five eBPF program types (tracepoint, kprobe, perf-event, XDP, and TC) and shares their in-kernel hookpoints. Rex only includes helpers for kernel interactions. helpers introduced due to contraints posed by the eBPF verifier (e.g., `bpf_loop`, `bpf_strtol`, and `bpf_strncmp`) are entirely excluded by Rex.

Kernel Crate

The Rex kernel crate is implemented in 3.5K lines of Rust code, among which 360 lines are unsafe Rust code. The kernel crate contains the following components:

- *Helper function interface* in Rex is implemented on top of eBPF helpers, with wrapping code that allows Rex extensions to invoke helpers with safe Rust types.
- *Kernel data-type bindings* are generated for the extension to access kernel data types defined in C. Rex uses rust-bindgen [212] to automatically generate kernel bindings and integrates it into the build process of extensions. Rex programs need to be rebuilt for each kernel they target to account for ABI differences in kernel data types.

- *Program context* in Rex is wrapped in a Rust struct, which marks the context as private and implements getter methods for its public fields.

Kernel Support

Rex implements the extension load code and the runtime in the kernel in 2.2K lines of C code on vanilla Linux. To load an extension, the kernel parses the ELF executable of the extension and locates all the `LOAD` segments in the executable. It then allocates new pages and maps the `LOAD` segments into the kernel address space based on the size and permissions of the segments. The load function is responsible for fixups on the program code to resolve referenced kernel helpers and eBPF maps. The Rex runtime in the kernel consists of the stack unwinding mechanisms (Section 4.4.5), support for dedicated kernel stack (Section 4.4.6) and termination (Section 4.4.7).

Compiler Support

Rex implements a compiler pass for Rex-specific compile-time instrumentations on the stack (Section 4.4.6). We take advantage of Rust's use of LLVM [213] as its default code generation backend and implement the pass in LLVM. A Rex-specific compiler switch is also added to the Rust compiler frontend (rustc [191]) to gate the Rex compiler pass.

4.6 EVALUATION

We evaluate Rex in terms of its usability and performance (with both macro and micro benchmarking).

4.6.1 Usability

Measuring usability is challenging. We evaluate Rex in two ways: (1) heuristic evaluation on whether it saves workarounds to the language-verifier gap, and (2) our dogfooding experience of using Rex to implement a large, complex extension (BMC [12]). Overall, we find that Rex enables developers to write simpler and cleaner code.

Eliminating Workarounds

Since Rex introduces no language-verifier gap, none of the workarounds in Section 4.2 is needed in writing Rex extensions.

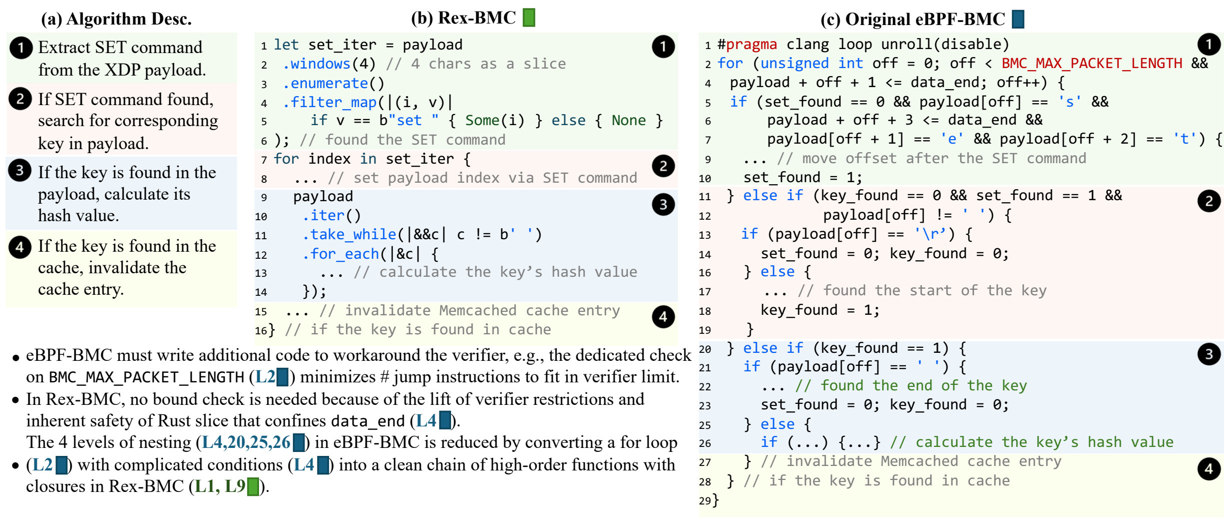
- Rex extensions have no limit on program size and complexity. There is no need to artificially refactor extension programs into smaller or simpler ones (Section 4.2.1).
- There is no need to artificially make Rex extensions verifier-friendly (Section 4.2.1). In fact, by decoupling static analysis from the kernel, Rex can enable new analysis (e.g., by allowing compilers to optimize for extra analysis/verification [214]).
- For the same reason, developers no longer need to tweak code to assist verification (Section 4.2.1).
- Developers no longer need to manage different verifier bugs across kernel versions (Section 4.2.1). The Rust compiler can have bugs and break safety guarantees, but it is arguably easier to upgrade than the kernel for fixes.
- Rex enables developers to use rich builtin intrinsics defined by the Rex toolchain without reinventing wheels (Section 4.2.1).

Case Study: Rex-BMC

We rewrite BMC [12] as a Rex kernel extension (Rex-BMC), which was originally written in eBPF extensions (eBPF-BMC). Rex-BMC is not a line-by-line translation of eBPF-BMC, because Rex provides more friendly programming experience (e.g., no need to split programs due to the verifier limit; see Section 4.2.1). In this section, we discuss Rex-BMC from the usability perspective and measure its performance in Section 4.6.2.

Our experience shows that Rex enables cleaner and simpler extension code, compared to eBPF. Essentially, Rex enables us to focus on key program logic without the overhead of passing the verifier. For example, we no longer need to divide code into in parts, add auxiliary code to help the verifier, dealing with tail calls and state transfer, etc. In addition, we can directly use Rust's builtin language features and libraries (e.g., iterators and closures). As one metric, Rex-BMC is written in 326 lines of Rust code. In comparison, eBPF-BMC is written in 513 lines of C code (splitting into seven programs).

Figure 4.7 compares the code snippets of eBPF-BMC and Rex-BMC that implement cache invalidation, respectively, as a qualitative example. The checks in eBPF-BMC code, required by the eBPF verifier, including these for offset and `data_end` limits, are now being enforced via the inherent language features of Rust, such as slices with bound checks in Rex (L2 and L10). The check on `BMC_MAX_PACKET_LENGTH`, which serves as a constraint to minimize the



(a) Algorithm Desc.

1. Extract SET command from the XDP payload.
2. If SET command found, search for corresponding key in payload.
3. If the key is found in the payload, calculate its hash value.
4. If the key is found in the cache, invalidate the cache entry.

(b) Rex-BMC 🟩

```
 1 let set_iter = payload
 2  .windows(4) // 4 chars as a slice
 3  .enumerate()
 4  .filter_map(|(i, v)|
 5     if v == b"set " { Some(i) } else { None }
 6 ); // found the SET command
 7 for index in set_iter {
 8   ... // set payload index via SET command
 9   payload
10    .iter()
11    .take_while(|&&c| c != b' ')
12    .for_each(|&c| {
13      ... // calculate the key’s hash value
14    });
15   ... // invalidate Memcached cache entry
16 } // if the key is found in cache
```

(c) Original eBPF-BMC 🟦

```
 1 #pragma clang loop unroll(disable)
 2 for (unsigned int off = 0; off < BMC_MAX_PACKET_LENGTH &&
 4   payload + off + 1 <= data_end; off++) {
 5   if (set_found == 0 && payload[off] == 's' &&
 6        payload + off + 3 <= data_end &&
 7        payload[off + 1] == 'e' && payload[off + 2] == 't') {
 9     ... // move offset after the SET command
10     set_found = 1;
11   } else if (key_found == 0 && set_found == 1 &&
12                payload[off] != ' ') {
13     if (payload[off] == '\r’) {
14       set_found = 0; key_found = 0;
16     } else {
17       ... // found the start of the key
18       key_found = 1;
19     }
20   } else if (key_found == 1) {
21     if (payload[off] == ' ') {
22       ... // found the end of the key
23       set_found = 0; key_found = 0;
25     } else {
26       if (...) {...} // calculate the key’s hash value
27     } // invalidate Memcached cache entry
28   } // if the key is found in cache
29 }
```

- eBPF-BMC must write additional code to workaround the verifier, e.g., the dedicated check on `BMC_MAX_PACKET_LENGTH` (**L2** 🟦) minimizes # jump instructions to fit in verifier limit.
- In Rex-BMC, no bound check is needed because of the lift of verifier restrictions and inherent safety of Rust slice that confines `data_end` (**L4** 🟦).
The 4 levels of nesting (**L4,20,25,26** 🟦) in eBPF-BMC is reduced by converting a for loop
- (**L2** 🟦) with complicated conditions (**L4** 🟦) into a clean chain of high-order functions with closures in Rex-BMC (**L1, L9** 🟩).

Figure 4.7: Cache invalidation implementation of Rex-BMC and eBPF-BMC; Rex leads to cleaner, simpler code.

number of jump instructions to circumvent the eBPF verifier, is no longer needed. Other checks for identified SET commands and loops states can be implemented with built-in functions and closures in an easy and clean way (L4–L6 and L11).

Moreover, with the elimination of program size and complexity limits in Rex-BMC, developers no longer have to save the computation state in a map across tail calls, which leads to clearer and more efficient implementation.

Note that the usability benefit does not come from the expressiveness difference between Rust and C, but from the closing of language-verifier gap via Rex. Evidently, the cleaner code of Rex-BMC would fail the verifier if it were to be compiled into eBPF (e.g., via Aya [164]): the compiler is unable to generate verifier-friendly code for convenient language features such as slices, and the verifier complexity limits will always be an issue. Rex allows us to fully leverage Rust's expressiveness without being constrained by verification issues.

Classroom Experience

We used Rex to develop two projects for CS 423 (Operating System Design) at the CS department in University of Illinois Urbana-Champaign in Fall 2022, 2023, and 2024. The students were asked to implement a packet filter and two kprobe programs using Rex. In total, 67 students attempted one of the two projects, and 32 correctly implemented the Rex extension programs that pass all of our grader tests; 65 students implemented working Rex extensions with partial functionality (earning partial credit). As OS learners, students found Rex easy to use and program with a low learning curve over Rust programming.

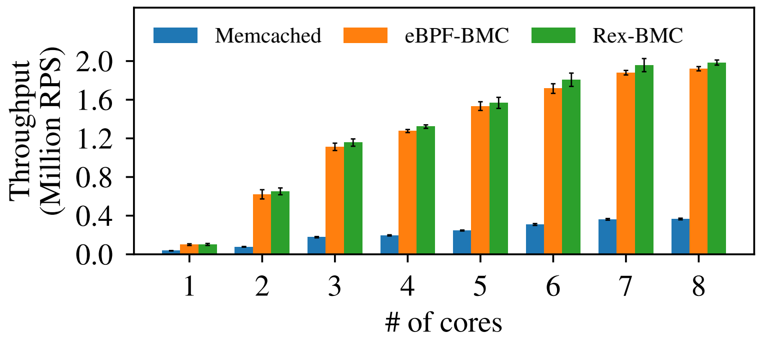


Figure 4.8: Throughput of Memcached, with eBPF-BMC, and with Rex-BMC under different number of cores.

With positive feedback, we are evolving the projects to larger, more complex Rex extension programs.

4.6.2 Macro benchmark

Rex's usability benefits do not come with a performance cost. We show that Rex delivers comparable performance as eBPF. Rex-BMC achieves a throughput of 1.98M requests per second (RPS) on 8 cores, which is slightly higher than eBPF-BMC (1.92M).

Our setup consists of a server machine and a client machine. The server machine runs the Rex custom kernel based on Linux v6.11.0 on an AMD EPYC 7551P 32-Core processor with 112 GB memory without SMT and Turbo. The client machine runs a vanilla v6.11.0 Linux kernel on an AMD Ryzen 9 9950X processor with 96 GB memory. Both machines are equipped with Mellanox ConnectX-3 Pro 40GbE NICs and are connected back-to-back using a single port.

We evaluate the throughput of (1) Memcached which binds multiple UDP sockets to the same port [12], (2) Memcached with eBPF-BMC, and (3) Memcached with Rex-BMC. For each setup, we vary the number of CPU cores for Memcached server and NIC IRQs and pin one Memcached thread onto each available core. We use the same workloads as in BMC [12], albeit with a smaller number of Memcached keys.

Figure 4.8 shows the throughput of the three setups under different numbers of CPU cores. Memcached processes all requests in userspace with the overhead of the kernel network stack, achieving only 37K RPS on a single core and 365K RPS on 8 cores. Both eBPF-BMC and Rex-BMC achieve a much higher throughput as they process a large fraction of requests at NIC driver level without going through the kernel network stack. With 8 cores, eBPF-BMC and Rex-BMC achieve a throughput of 1.92M and 1.98M, and a performance benefit of 5.26x and 5.43x, respectively. The slight performance improvement over eBPF is attributable to the elimination of overheads of tail calls and associated state-passing via maps, along with optimizations in the rustc frontend and x86 backend, despite the additional runtime checks.

4.6.3 Micro benchmark

Several of Rex's designs could introduce overheads, despite invisible in the Rex-BMC evaluation. We use microbenchmarks specifically designed to stress our design and measure overheads. We show that overheads exist in some pessimistic cases, but have negligible impact in real-world scenarios. All experiments are performed on the same machine that acts as the server in the Rex-BMC experiments (Section 4.6.2).

Setup and Teardown

Entering and exiting a Rex program requires Rex-specific operations (Figure 4.6). Rex's use of a dedicated stack requires saving the stack pointer and setting the new stack and frame pointer to the dedicated stack (and restoring to the saved values after the extension exits). Rex also needs to set up the per-CPU state used by its termination mechanism (Section 4.4.7). In total, these operations add eight instructions on the execution path in Rex. To measure the overhead, we implement an empty extension program in both eBPF and Rex and record their execution time (including the program dispatcher). As shown in Table 4.2, the measured execution time of the empty Rex and eBPF programs only differ in around a nanosecond on average.

Exception Handling

Rex's safe cleanup for exception handling requires recording allocated resource at runtime (Section 4.4.5), which, compared to eBPF, adds overhead. We measure the overhead using a program that acquires and then immediately releases an eBPF spinlock. Since the acquired spinlock needs to be released upon Rust panics, Rex's cleanup mechanism records it in its per-CPU buffer. Additionally, Rex sets up a per-CPU state flag to indicate execution of a helper function (Section 4.4.7). The program is implemented in both eBPF and Rex and the time used to acquire and release the spinlocks are measured. Table 4.2 shows that the runtime difference between eBPF and Rex is roughly 50 nanoseconds.

Table 4.2: Time to execute an empty extension program and to acquire and release a spinlock in eBPF and Rex (nanosecond)

Extension	**Empty prog runtime**	**Spinlock runtime**
eBPF	42.1 ± 4.1 ns	130.4 ± 20.3 ns
Rex	42.6 ± 5.8 ns	183.1 ± 27.5 ns

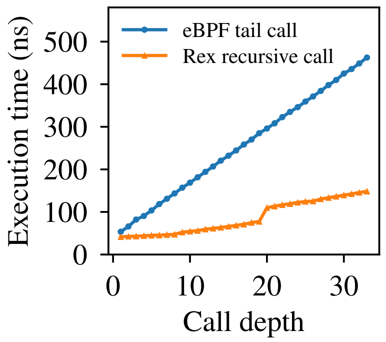


Figure 4.9: eBPF tail call and Rex recursive call time

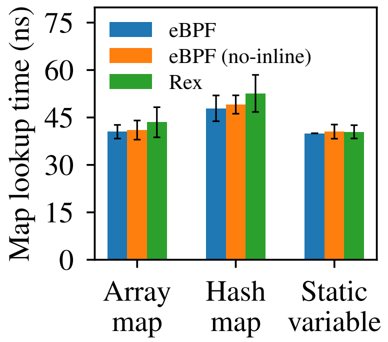


Figure 4.10: Map lookup time under various setups

Stack Check

Stack checks are added before function calls in Rex extensions that contain indirect or recursive calls (Section 4.4.6). We implement recursive extension programs in both eBPF and Rex to measure the overhead. The recursive function calls itself for a controlled number of times. In Rex, we pass the call depth as the argument to the recursive function; since eBPF does not support recursive functions, we use eBPF tail calls to implement the logic—since it is inconvenient to pass arguments to tail-called programs (Section 4.2.1), we use a static variable to set the call depth. Figure 4.9 plots execution time of the recursive programs with call depths from 1 to 33 (eBPF cannot do more than 33 tail calls). Rex is roughly 3x faster than eBPF. The overhead of eBPF is due to runtime check on tail-call count limit and accessing the static variable, which is a map in eBPF (not a register in normal calls).

Map Access

Map access in Rex is expected to have more overhead than in eBPF. First, Rex implements wrapping code to enforce safety of helper function calls (Section 4.5). Moreover, the eBPF JIT compiler inlines the helper function for map lookup at load time as a performance optimization; however, inlining is not available in Rex (no JIT in Rex). We measure map lookup time of Rex, compared with eBPF with and without inlining, including array map, hash map, and static variable. In eBPF, static variables are converted into maps; we use a static Rust atomic variable in Rex, as the counterpart of a static variable map in eBPF. Figure 4.10 shows the lookup time of different maps in eBPF and Rex, respectively. We find that inlining map lookups in eBPF are ∼0.5ns faster on array maps and ∼1.2ns faster on hash maps. An additional slowdown of 2ns–4ns is present in Rex over non-inlined eBPF, due to the wrapping code. Static variables in eBPF are always accessed via direct load without invoking a helper. Hence, their access latency is almost the same to accessing Rust atomic variables.

4.7 DISCUSSION

Verification without Language-verifier Gaps

Rex currently uses language features of Rust to ensure safety of kernel extensions. This approach defers the checking of some safety properties to the runtime (e.g., termination, integer errors). It may be possible to minimize the amount of runtime errors by incorporating Rust-based verification techniques, e.g., ensuring freedom of panics [187, 188, 189, 215, 216]. Certainly, push-button verification techniques that use symbolic execution such as PanicCheck [215] are likely to re-introduce the language-verifier gap. We suspect that using verification techinques that combine proofs and programming [187, 188, 217, 218], such as Verus for Rust may allow Rex to reduce runtime errors *without* the language-verifier gap.

Dynamic Memory Allocation

eBPF has recently supported dynamic allocation [219] that allows extension programs to request memory from the kernel using allocation kfuncs [220]. Rex currently does not support dynamic memory allocation. We plan to integrate memory allocation [195] of Rust with the eBPF all-context allocator [221], granting Rex dynamic allocation. Dynamic allocation enhances programmability of extension programs and opens up more advanced use cases [7]. It also makes more components of the Rust standard library available, notably the collection and smart pointer types with automatic memory management.

Kernel Crate Maintenance

The Rex kernel crate inevitably needs to use unsafe Rust, as it directly interacts with kernel functions and variables. As a principle, unsafe Rust code must not be used for escaping safety checks but only when it is the last resort (mostly for foreign function interface, FFI). This keeps the scope of unsafe Rust at its minimum—the Rex kernel crate only leverages unsafe Rust necessary for FFI interaction and contributes to about 10% (360 lines) of kernel crate code. As unsafe code is isolated from extension programs and managed at a central location by trusted maintainers, we are not particularly concerned about its maintainability.

4.8 SUMMARY

In this chapter, we advocate for a new kernel extension design that leverages language-based safety combined with a light-weight runtime, to avoid the challenges of the language-verifier gap in existing static verification schemes. We build Rex, a new kernel extension framework that closes the language-verifier gap. We believe that closing the gap is essential to programming experience and maintainability of kernel extensions, especially those that embody large, complex programs for advanced features. Rex provides a solution that allows kernel extensions to be developed and maintained in a high-level language, while providing desired safety guarantees as the existing framework like eBPF.

CHAPTER 5: RELATED WORK

This dissertation identifies and addresses the important expressiveness, programmability, and efficiency problems on safe kernel extensions with the work of Seccomp-eBPF, Rex, Uno-kprobe, and eKCFI in their respective contexts. In this chapter, we discuss previous works related to each of the problems and the corresponding systems: Section 5.1 discusses general system call filtering mechanisms, as well as this specific use case of kernel extensions; Section 5.2.1 reviews previous works on program tracing and instrumentation, both with and without the use of traps; Section 5.2.2 dives into different CFI policies from advancing analysis techniques, as well as the current state of CFI mechanisms in kernel space; lastly, Section 5.3 revisits works that improve the ease of program for existing eBPF and others that focus on the safety of kernel extensions in general.

5.1 EXPRESSIVENESS

System Call Filtering

Prior work has studied system call filtering (aka interposition) techniques for protecting the shared OS kernel against untrusted applications [36, 37, 84, 91, 222, 223, 224, 225, 226, 227, 228, 229, 230, 231]. Early techniques rely on userspace agents (e.g., based on `ptrace`) that check user-specified policies to decide which system calls to allow or deny, in the same vein as Seccomp Notifier (see Section 2.1.2). 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 cBPF 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 [80, 81]. However, our discussions with the community both on the kernel mailing list [69] and at the Linux Plumbers Conference [70] tell that it is still very controversial.

One common concern is lacking concrete use cases, as exemplified by the maintainer's response—"*What's the reason for adding eBPF support? Seccomp shouldn't need it... I'd rather stick with cBPF until we have an overwhelmingly good reason to use eBPF...*" [232]. One reason is that early patches [80, 81] 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 (Section 2.2). 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—"*You are going to have a very hard time convincing the Seccomp maintainers to let any of these mechanism interact with Seccomp...*" [233], and (2) potential vulnerabilities in the eBPF subsystem—"*This is the blocker as far as I'm concerned: there is no story for unprivileged eBPF.*" [234]. The quotes are from our discussions with Linux maintainers and engineers. In our design, the security of Seccomp-eBPF can be systematically reduced to that of Seccomp and eBPF. 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 (Section 2.5) which is useful for container environments.

Other eBPF Use Cases

In recent years, eBPF has evolved from simple use cases like packet filtering [10, 11] into a general-purpose kernel extension language and programming framework that enables many innovative projects [12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 235, 236, 237]. Different from most of the eBPF use cases, Seccomp needs to support unprivileged use cases (Section 2.3.1). Therefore, security is a first-class design principle of Seccomp-eBPF.

5.2 EFFICIENCY

5.2.1 Kernel probes

SystemTap [238] and DTrace [239] both use trap-based probing mechanisms. Similarly, Ptrace [240] uses traps to provide userspace applications with a mechanism to hook onto processes. Another trap-based probing mechanism is the Xenprobes [241] for probing guest kernels. Mechanisms for secure active monitoring [242], dynamic operating systems monitoring [243], memory introspection [244], and intrusion detection in the kernel [245] use forms of trap-based probing. Our work offers a principled solution to optimize probes with handlers sharing the same address space, but would require a different design for probes requiring a world-switch.

Unlike trap-based techniques, many instrumentation probes utilize trampolines instead of breakpoints. The Ftrace function tracer [246] in the Linux kernel utilizes compiler-placed `nops` at the beginning of functions and rewrites them dynamically to jump to trampolines. A similar mechanism is also proposed for enforcing kernel CFI due to its performance benefit [63]. Both mechanisms only work on probing specific points in the kernel and are far from being universal, unlike our work. At the same time, DynamoRIO [247], DynInst [248], and Intel Pin [249] are trampoline-based tools where some of them use emulation, which might benefit from our work and would be an interesting future work.

5.2.2 KCFI

CFI Policies

CFI thwarts control flow hijack attacks by checking indirect control flow transfer targets against valid targets. A CFI policy consists of a Control Flow Graph (CFG) and an enforcement policy. The CFG encodes control flow information of the target program, including locations of indirect control transfers, valid control transfer targets, and their program context. The CFG can be generated with static or dynamic analysis [48, 49, 50, 51, 52, 55, 250]. Some CFI also considers runtime context [75, 251, 252, 253, 254, 255, 256, 257]. The enforcement policy determines when (temporal) and where (spatial) to apply CFI checks and the actions upon violations.

Traditional CFI mechanisms like KCoFI [48] struggle with precision, because their CFGs contain large sets of potential control flow transfer targets for a given transfer [57, 58, 258, 259]. Recently, the advancement of static analysis on indirect control flow targets based on type information [59, 60], taint tracking [49], pointer analysis [50, 61], and others [62] enables more precise CFI policies to raise the bar on attackers.

Existing Kernel CFI Mechanisms

Existing KCFI mechanisms only consider static, compile-time information [52, 55] in its policy including both CFG and runtime enforcement. Specifically, the *de facto* KCFI for Linux, referred to as LLVM-KCFI, uses a type-based policy—an indirect call is allowed only if the type of the actual target matches that of the function pointer. Specifically, LLVM-

KCFI generates a hash value for each function with its address taken based on its prototype. This hash value is compared by instrumented code at each indirect call site with another hash value generated from the type of the function pointer. If the value matches, the indirect call is allowed; otherwise, the execution triggers a kernel oops.

With LLVM-KCFI, it is nontrivial to use a different KCFI policy. First, it is hard to leverage advanced static analysis techniques to generate more precise CFGs or to consider runtime context. Moreover, after a KCFI-enforced kernel is deployed, it is hard to change the enforcement policy, because the enforcement mechanism is compiled into the kernel. For example, it is even hard to dynamically enable or disable KCFI enforcement—doing so requires recompiling and rebooting the kernel, which can be disruptive and time-consuming.

eKCFI addresses the above limitations by enabling system administrators to program KCFI policies using eBPF.

5.3 PROGRAMMABILITY

Improving eBPF

Recent work is making active progress to improve the correctness and security of the eBPF infrastructure, including fuzzing and bug finding [260, 261, 262, 263], formal verification [137, 140, 172, 264], sandboxing [158, 159, 173], and integrating with hardware protection mechanisms [265, 266]. eBPF's design, which relies on an in-kernel static verifier for extension safety, inevitably creates the language-verifier gap (Section 4.2). In contrast, Rex provides an alternative to develop and maintain large, complex kernel extensions directly with high-level language safety, avoiding the language-verifier gap.

Other Frameworks

The idea of building safe OS components using safe languages was proposed by SPIN [4] and revisited by Singularity [267], Tock [179], and a few recent discussions [2, 77, 268]. However, adopting them in practice is challenging as they are based on clean-slate OS designs. Rex develops a practical kernel extension framework for Linux, taking the opportunity of recent support of Rust as a safe language for OS code. It addresses the key challenges of enforcing safe code only, interfacing with unsafe C code, and providing safety guarantees beyond language-based safety.

KFlex [7] is a recent kernel extension framework built on top of eBPF. KFlex aims to improve the flexibility of eBPF to let developers express diverse functionality in extensions.

It employs an efficient runtime by co-designing it with the existing eBPF verifier: (1) its Software Fault Isolation (SFI) elides checks already done by the verifier for efficiency, and (2) its termination mechanism uses the verifier to statically compute the kernel resources acquired by the extension. Rex made the same design choice as KFlex to use a lightweight runtime for safety properties that are hard to check statically. Unlike KFlex, which is co-designed with the eBPF verifier, Rex eliminates the verifier to close the language-verifier gap.

BCF [8] is a recent proposal to enhance eBPF's in-kernel verification with help from user space, asking for proof when the verifier fails to reason about certain program properties. The idea echoes proof-carrying code [269] which asks a program to attach a formal proof that its code obeys the safety policy. BCF leverages the eBPF verifier's range analysis and symbolic execution for proof generation but still requires developers to specify safety conditions to aid the generation. Its uses of the verifier still lead to the language-verifier gap.

Rust for OS Kernels

Rust has been embraced by modern OSes [1, 270] as practical language which leads to safer code. Recent work shows the promises to build new OS kernels using Rust [156, 157, 179, 268]. We claim no novelty of using Rust as a language. In fact, a safe language alone does not lead to system safety, as exemplified by Rust kernel modules [3]. Rex shows an example of how to build upon language-based safety to enable and enforce safe kernel extension programs.

CHAPTER 6: CONCLUSION

This dissertation identifies major limitations of existing kernel extension frameworks in the aspect of expressiveness, efficiency, and programmability as a result of improper design tradeoffs and aims to solve them by providing designs that implement the correct tradeoff in our perspective. We hope our work in this dissertation sheds some light on how different aspects of safe kernel extension design may be balanced in the future.

This dissertation first addresses the lack of expressiveness in the context of system call filtering, namely Seccomp on Linux, due to the existing design overly emphasizing the safety and security on the safety axis of the extension design. We introduce a new, programmable system call filtering framework that leverages the enhanced expressiveness from eBPF extensions in Seccomp. The design of Seccomp-eBPF strictly adheres to the existing security model of Seccomp and eBPF and can securely support unprivileged use cases. We demonstrate that Seccomp-eBPF can both improve existing system call filtering policies and support new policies requiring higher filter expressiveness.

This dissertation then seeks to solve the efficiency limitation of kernel extensions in their use cases of kernel probing and KCFI. In both cases, the limitation is caused by the design of the attachment mechanisms that aims for simplicity on the extensibility axis of extension design and sacrifices efficiency. For the kernel probing use case, we identify the trap instructions used by existing kernel probe implementations create major performance bottlenecks. This dissertation, therefore, presents the design of a universally trapless kernel probe mechanism based on strategically placed *nops*. Uno-kprobe, an implementation of this design on Linux's kprobe, achieves a 10x single-probe performance boost over existing trap-based kprobes and covers 96% of kernel code with fast kernel probes. For the KCFI use case, we point out that the existing eBPF infrastructure is fundamentally limited in providing the needed level of efficiency, scalability, and coverage for a KCFI mechanism. We present eKCFI, an eBPF-based KCFI framework that employs new instrumentation and attachment mechanisms to provide efficient kernel code instrumentation, scalable program attachment, and complete coverage of forward-edge control flow transfers. We show that eKCFI can support flexible KCFI policies while incurring only reasonable performance overhead.

Lastly, this dissertation highlights the programmability limitation of the existing eBPF kernel extension mechanism on Linux, where its static verification-based safety checks make eBPF extensions hard to program, as the verifier frequently rejects correct and safe programs. We point out that this limitation is a result of the language-verifier gap that fundamentally exists in verification-based safety solutions that give in the programmability aspect on the safety axis of extension design. We propose a new design of safe kernel extension frameworks that utilizes language-based safety combined with a lightweight extralingual runtime to provide safety guarantees. We realize our design in the Rex kernel extension framework with the Rust language and showcase that Rex can eliminate the cumbersome verifier workarounds and effectively close the language-verifier gap, improving programmability without additional performance overhead.

APPENDIX A: DEEP ARGUMENT INPECTION IN SECCOMP-EBPF

We discuss our helper function implementations that support checking argument values without being susceptible to TOCTTOU vulnerabilities [108, 113]. The key principle is to disallow userspace applications to modify the argument values as long as the values are used in a system call.

There are two basic approaches to implement such helpers without additional hardware support: (1) making the data page write-protected or inaccessible from the user space by modifying protection bits in the page tables (e.g., RW bit and US bit), or (2) copying the argument values to a protected memory region that can only be modified by the kernel.

For Seccomp-eBPF, we have implemented both approaches. The first approach, called DPTI [113], requires modifying protection bits in the page tables and flushing the corresponding TLB entry from all necessary CPU cores. Moreover, it requires modifications to the page fault handler. The reason for that is that the data page can be accessed by concurrent threads for data on the page that is unrelated to the system call. In the case of the modified RW bit, a write access by a concurrent thread causes a page fault while reads are permitted. In the case of the modified US bit, both read and write accesses cause a page fault. Hence, the page fault handler has to temporarily stall threads that perform a normally legitimate access to such modified pages.

For the second approach, we copy the argument values to dedicated kernel pages allocated for each thread, before entering Seccomp. The kernel page is mapped to a userspace address, which is read-only to prevent TOCTTOU vulnerabilities. We disable `VM_MAYWRITE` of the page to prevent the user space from gaining the write permission (e.g., via `mprotect`). Since the page is mapped in the user space, the kernel can maintain its original workflow of copying system call argument values from the user space. So, we can directly use the existing user-memory reading helpers in the eBPF filter to access the argument values. Note that it needs to know how to copy the argument values for different system calls, i.e., it needs to know the argument semantics, such as the size of the object and the structures for nested pointers. When a system call is invoked, an in-kernel hook is triggered to copy the argument values based on the semantic of the particular system call. The hook is only enabled when an eBPF filter that uses the user-memory reading helpers are attached to the current thread.

REFERENCES

[1] "Rust for Linux," https://rust-for-linux.com/.

[2] S. Miller, K. Zhang, D. Zhuo, S. Xu, A. Krishnamurthy, and T. Anderson, "Practical Safe Linux Kernel Extensibility," in *Proceedings of the 17th ACM Workshop on Hot Topics in Operating Systems (HotOS'21)*, May 2021.

[3] J. Corbet, "Rust-for-Linux developer Wedson Almeida Filho drops out," https://lwn.net/Articles/987635/. (Aug. 2024).

[4] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers, "Extensibility, Safety and Performance in the SPIN Operating System," in *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*, Dec. 1995.

[5] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith, "Dealing With Disaster: Surviving Misbehaved Kernel Extensions," in *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI'96)*, Oct. 1996.

[6] "BPF and XDP Reference Guide," https://docs.cilium.io/en/latest/bpf/index.html.

[7] K. K. Dwivedi, R. Iyer, and S. Kashyap, "Fast, Flexible, and Practical Kernel Extensions," in *Proceedings of the 30th ACM Symposium on Operating Systems Principles (SOSP'24)*, Nov. 2024.

[8] H. Sun and Z. Su, "Lazy Abstraction Refinement with Proof. In *Linux Plumbers Conference (LPC'24)*," https://lpc.events/event/18/contributions/1939/. (Sept. 2024).

[9] "eBPF implementation that runs on top of Windows," https://github.com/microsoft/ebpf-for-windows.

[10] J. C. Mogul, R. F. Rashid, and M. J. Accett, "The Packet Filter: An Efficient Mechanism for User-level Network Code," in *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP'87)*, Nov. 1987.

[11] S. McCanne and V. Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture," in *Proceedings of the 1993 Winter USENIX Conference*, Jan. 1993.

[12] Y. Ghigoff, J. Sopena, K. Lazri, A. Blin, and G. Muller, "BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing," in *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI'21)*, Apr. 2021.

[13] Y. Zhong, H. Li, Y. J. Wu, I. Zarkadas, J. Tao, E. Mesterhazy, M. Makris, J. Yang, A. Tai, R. Stutsman, and A. Cidon, "XRP: In-Kernel Storage Functions with eBPF," in *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)*, July 2022.

[14] X. Cao, S. Patel, S. Y. Lim, X. Han, and T. Pasquier, "FetchBPF: Customizable Prefetching Policies in Linux with eBPF," in *Proceedings of the 2024 USENIX Annual Technical Conference (USENIX ATC'24)*, July 2024.

[15] Z. Yang, Y. Lu, X. Liao, Y. Chen, J. Li, S. He, and J. Shu, "λ-IO: A Unified IO Stack for Computational Storage," in *Proceedings of the 21st USENIX Conference on File and Storage Technologies (FAST'23)*, Feb. 2023.

[16] Y. Zhou, Z. Wang, S. Dharanipragada, and M. Yu, "Electrode: Accelerating Distributed Protocols with eBPF," in *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI'23)*, Apr. 2023.

[17] Y. Zhou, X. Xiang, M. Kiley, S. Dharanipragada, and M. Yu, "DINT: Fast In-Kernel Distributed Transactions with eBPF," in *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI'24)*, Apr. 2024.

[18] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The eXpress data path: fast programmable packet processing in the operating system kernel," in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT'18)*, Dec. 2018.

[19] J. T. Humphries, N. Natu, A. Chaugule, O. Weisse, B. Rhoden, J. Don, L. Rizzo, O. Rombakh, P. Turner, and C. Kozyrakis, "ghOSt: Fast & Flexible User-Space Delegation of Linux Scheduling," in *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP'21)*, Oct. 2021.

[20] K. Mores, S. Psomadakis, and G. Goumas, "eBPF-mm: Userspace-guided memory management in Linux with eBPF," *arXiv:2409.11220*, Sep. 2024.

[21] "Extensible Scheduler Class," https://docs.kernel.org/next/scheduler/sched-ext.html.

[22] J. Edge, "A seccomp overview," https://lwn.net/Articles/656307/, Sep. 2015.

[23] "Seccomp security profiles for Docker," https://docs.docker.com/engine/security/seccomp/.

[24] "gVisor: Container Runtime Sandbox," https://github.com/google/gvisor/blob/master/runsc/boot/filter/config.go.

[25] "Firecracker Design," https://github.com/firecracker-microvm/firecracker/blob/master/docs/design.md.

[26] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight Virtualization for Serverless Applications," in *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*, Feb. 2020.

[27] V. Narayanan, A. Balasubramanian, C. Jacobsen, S. Spall, S. Bauer, M. Quigley, A. Hussain, A. Younis, J. Shen, M. Bhattacharyya, and A. Burtsev, "LXDs: Towards isolation of kernel subsystems," in *Proc. of USENIX Annual Technical Conf.*, Renton, WA, July 2019.

[28] Rtk docs, "Seccomp Isolators Guide," https://coreos.com/rkt/docs/latest/seccomp-guide.html.

[29] sylabs/singularity, "Singularity: Application containers for Linux," https://github.com/sylabs/singularity/blob/master/etc/seccomp-profiles/default.json.

[30] Kubernetes Documentation, "Configure a Security Context for a Pod or Container," https://kubernetes.io/docs/tasks/configure-pod-container/security-context/, July 2019.

[31] "Linux Seccomp Support in Mesos Containerizer," http://mesos.apache.org/documentation/latest/isolators/linux-seccomp/.

[32] J. Corbet, "Systemd gets seccomp filter support," https://lwn.net/Articles/507067/.

[33] "Sandboxed API," https://github.com/google/sandboxed-api.

[34] J. Corbet, "BPF: the universal in-kernel virtual machine," https://lwn.net/Articles/599755/, May 2014.

[35] C. Brauner, "The Seccomp Notifier – New Frontiers in Unprivileged Container Development," https://brauner.github.io/2020/07/23/seccomp-notify.html, July 2020.

[36] D. A. Wagner, "Janus: an Approach for Confinement of Untrusted Applications," EECS Department, University of California, Berkeley, Tech. Rep. UCB/CSD-99-1056, 2016.

[37] T. Garfinkel, B. Pfaff, and M. Rosenblum, "Ostia: A Delegating Architecture for Secure System Call Interposition," in *Proceedings of the 2004 Network and Distributed System Security Symposium (NDSS'04)*, Feb. 2004.

[38] T. Garfinkel, "Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools," in *Proceedings of the 2004 Network and Distributed System Security Symposium (NDSS'04)*, Feb. 2003.

[39] A. Bhattacharyya, U. Tesic, and M. Payer, "Midas: Systematic Kernel TOCTTOU Protection," in *Proceedings of the 31st USENIX Security Symposium (USENIX Security'22)*, Aug. 2022.

[40] M. Payer and T. R. Gross, "Protecting Applications Against TOCTTOU Races by User-Space Caching of File Metadata," in *Proceedings of the 8th Annual International Conference on Virtual Execution Environments (VEE'12)*, Mar. 2012.

[41] "BPF Compiler Collection (BCC)," https://github.com/iovisor/bcc.

[42] "bpftrace," https://github.com/bpftrace/bpftrace.

[43] "Ved-ebpf: Kernel exploit and rootkit detection using ebpf," https://github.com/hardenedvault/ved-ebpf.

[44] G. Fournier, "Return to Sender - Detecting Kernel Exploits with eBPF. In *Black Hat USA 2022*," https://i.blackhat.com/USA-22/Wednesday/US-22-Fournier-Return-To-Sender.pdf. (Aug. 2022).

[45] "System Analysis and Tuning Guide," https://documentation.suse.com/sles/15-SP3/html/SLES-all/book-tuning.html. (Mar. 2025).

[46] M. Hiramatsu, "The Enhancement of Kernel Probing — Kprobes Jump Optimization. In *Tracing Summit 2010*," https://tracingsummit.org/ts/2010/files/HiramatsuLinuxCon2010.pdf. (Aug. 2010).

[47] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-Flow Integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS'05)*, Nov. 2005.

[48] J. Criswell, N. Dautenhahn, and V. Adve, "KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels," in *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P'14)*, May 2014.

[49] X. Ge, N. Talele, M. Payer, and T. Jaeger, "Fine-Grained Control-Flow Integrity for Kernel Software," in *Proceedings of the 1st IEEE European Symposium on Security and Privacy (EuroS&P'16)*, Mar. 2016.

[50] J. Li, X. Tong, F. Zhang, and J. Ma, "Fine-CFI: Fine-Grained Control-Flow Integrity for Operating System Kernels," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 6, June 2018.

[51] J. Moreira, S. Rigo, M. Polychronakis, V. P. Kemerlis, and S. Killer, "DROP THE ROP Fine-grained Control-flow Integrity for the Linux Kernel. In *Black Hat Asia 2017*," https://www.blackhat.com/docs/asia-17/materials/asia-17-Moreira-Drop-The-Rop-Fine-Grained-Control-Flow-Integrity-For-The-Linux-Kernel-wp.pdf. (Mar. 2017).

[52] S. Tolvanen, "Control flow integrity in the android kernel," https://android-developers.googleblog.com/2018/10/control-flow-integrity-in-android-kernel.html. (Oct. 2018).

[53] "Pointer Authentication," https://github.com/apple/llvm-project/blob/a63a81bd9911f87a0b5dcd5bdd7ccdda7124af87/clang/docs/PointerAuthentication.rst.

[54] "Control Flow Guard," https://docs.microsoft.com/en-us/windows/win32/secbp/control-flow-guard.

[55] J. Corbet, "A new LLVM CFI implementation," https://lwn.net/Articles/898040/. (Jun. 2022).

[56] S. Tolvanen, "[PATCH v5 05/22] cfi: Switch to -fsanitize=kcfi," https://lore.kernel.org/all/20220908215504.3686827-6-samitolvanen@google.com/. (Sept. 2022).

[57] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-Flow Bending: On the Effectiveness of Control-Flow Integrity," in *Proceedings of the 24th USENIX Security Symposium (USENIX Security'15)*, Aug. 2015.

[58] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS'15)*, Oct. 2015.

[59] K. Lu, "Practical Program Modularization with Type-Based Dependence Analysis," in *Proceedings of the 44th IEEE Symposium on Security and Privacy (S&P'23)*, May 2023.

[60] K. Lu and H. Hu, "Where Does It Go? Refining Indirect-Call Targets with Multi-Layer Type Analysis," in *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS'19)*, Nov. 2019.

[61] S. H. Kim, C. Sun, D. Zeng, and G. Tan, "Refining indirect call targets at the binary level." in *Proceedings of the Network and Distributed System Security Symposium 2021 (NDSS'21)*, Feb. 2021.

[62] Q. Wu, Z. Gu, H. Jamjoom, and K. Lu, "Gnnic: Finding long-lost sibling functions with abstract similarity," in *Proceedings of the Network and Distributed System Security Symposium 2024 (NDSS'24)*, Feb. 2024.

[63] J. Jia, M. V. Le, S. Ahmed, D. Williams, and H. Jamjoom, "Practical and Flexible Kernel CFI Enforcement using eBPF," in *Proceedings of the 1st Workshop on EBPF and Kernel Extensions (eBPF'23)*, Sep. 2023.

[64] A. Maestretti and B. Gregg, "Security Monitoring with eBPF. In *BSides San Francisco 2017*," https://www.brendangregg.com/Slides/BSidesSF2017_BPF_security_monitoring.pdf. (Feb. 2017).

[65] "LSM BPF Programs," https://docs.kernel.org/bpf/prog_lsm.html.

[66] J. Edge, "BPF and security," https://lwn.net/Articles/946389/. (Oct. 2023).

[67] "Kernel Probes (Kprobes)," https://docs.kernel.org/trace/kprobes.html.

[68] B. Gregg, "Linux Extended BPF (eBPF) Tracing Tools," https://www.brendangregg.com/ebpf.html.

[69] Y. Zhu, "eBPF seccomp filters," https://lwn.net/Articles/855970/ (May. 2021).

[70] J. Jia, Y. Zhu, A. Arcangeli, H. Franke, T. Feldman-Fitzthum, C. Canella, D. Skarlatos, D. Gruss, D. Williams, and T. Xu, "Revisiting eBPF Seccomp Filters. In *Linux Plumbers Conference (LPC'22)*."

[71] J. Jia, Y. Zhu, D. Williams, A. Arcangeli, C. Canella, H. Franke, T. Feldman-Fitzthum, D. Skarlatos, D. Gruss, and T. Xu, "Programmable System Call Security with eBPF," *arXiv:2302.10366*, Feb. 2023.

[72] J. Jia, "[PATCH] x86/kprobes: fix incorrect return address calculation in kprobe_emulate_call_indirect," https://lore.kernel.org/all/20240102233345.385475-1-jinghao7@illinois.edu/. (Jan. 2024).

[73] J. Jia, "[PATCH v2 0/3] x86/kprobes: add exception opcode detector and boost more opcodes," https://lore.kernel.org/all/20240204031300.830475-1-jinghao7@illinois.edu/. (Feb. 2024).

[74] J. Jia, M. V. Le, S. Ahmed, D. Williams, H. Jamjoom, and T. Xu, "Fast (Trapless) Kernel Probes Everywhere," in *Proceedings of the 2024 USENIX Annual Technical Conference (USENIX ATC'24)*, July 2024.

[75] M. Khandaker, A. Naser, W. Liu, Z. Wang, Y. Zhou, and Y. Cheng, "Adaptive Call-Site Sensitive Control Flow Integrity," in *Proceedings of the 4th IEEE European Symposium on Security and Privacy (EuroS&P'19)*, June 2019.

[76] J. Jia, M. V. Le, S. Ahmed, D. Williams, H. Jamjoom, and T. Xu, "Advancing Kernel Control Flow Integrity with eBPF. In *Linux Plumbers Conference (LPC'23)*."

[77] J. Jia, R. Sahu, A. Oswald, D. Williams, M. V. Le, and T. Xu, "Kernel extension verification is untenable," in *Proceedings of the 19th ACM Workshop on Hot Topics in Operating Systems (HotOS'23)*, June 2023.

[78] J. Jia, R. Qin, M. Craun, E. Lukiyanov, A. Bansal, M. V. Le, H. Franke, H. Jamjoom, T. Xu, and D. Williams, "Safe and usable kernel extensions with Rax," *arXiv:2502.18832*, Feb. 2025.

[79] M. Kerrisk, "Using seccomp to limit the kernel attack surface. In *Linux Plumbers Conference (LPC'15)*."

[80] S. Dhillon, "eBPF Seccomp filters," https://lwn.net/Articles/747229/ (Feb. 2018).

[81] T. Hromatka, "[RFC PATCH] all: RFC - add support for ebpf," https://groups.google.com/g/libseccomp/c/pX6QkVF0F74/m/ZUJlwI5qAwAJ, (Feb. 2018).

[82] J. Edge, "A seccomp overview," https://lwn.net/Articles/656307/, Sep. 2015.

[83] H.-C. Kuo, K.-H. Chen, Y. Lu, D. Williams, S. Mohan, and T. Xu, "Verified Programs Can Party: Optimizing Kernel Extensions via Post-Verification Merging," in *Proceedings of the 17th European Conference on Computer Systems (EuroSys'22)*, Apr. 2022.

[84] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer, "A Secure Environment for Untrusted Helper Applications (Confining the Wily Hacker)," in *Proceedings of the 6th USENIX Security Symposium (USENIX Security '96)*, July 1996.

[85] "Podman: A tool for managing OCI containers and pods," https://podman.io/.

[86] "runc standard_init_linux.go," https://github.com/opencontainers/runc/blob/master/libcontainer/standard_init_linux.go#L161-L230.

[87] "Docker's default Seccomp profile," https://github.com/moby/moby/blob/master/profiles/seccomp/default.json.

[88] "Kubernetes's default crun profile," https://github.com/kubernetes-sigs/security-profiles-operator/blob/master/examples/baseprofile-crun.yaml.

[89] C. Canella, S. Dorn, D. Gruss, and M. Schwarz, "SFIP: Coarse-Grained Syscall-Flow-Integrity Protection in Modern Systems," *arXiv:2202.13716*, Feb. 2022.

[90] F. Maggi, M. Matteucci, and S. Zanero, "Detecting Intrusions through System Call Sequence and Argument Analysis," *IEEE Transactions on Dependable and Secure Computing (TDSC)*, vol. 7, no. 4, pp. 381–395, Oct. 2010.

[91] C. M. Linn, M. Rajagopalan, S. Baker, C. Collberg, S. K. Debray, and J. H. Hartman, "Protecting Against Unexpected System Calls," in *Proceedings of the 14th USENIX Security Symposium (USENIX Security '05)*, July 2005.

[92] Christina Warrender and Stephanie Forrest and Barak Pearlmutter, "Detecting Intrusions Using System Calls: Alternative Data Models," in *Proceedings of the 1999 IEEE Symposium on Security and Privacy (S&P'99)*, May 1999.

[93] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A Sense of Self for Unix Processes," in *Proceedings of the 1996 IEEE Symposium on Security and Privacy (S&P'96)*, May 1996.

[94] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong, "Anomaly Detection Using Call Stack Information," in *Proceedings of the 2003 IEEE Symposium on Security and Privacy (S&P'03)*, May 2003.

[95] A. K. Ghosh and A. Schwartzbard, "A Study in Using Neural Networks for Anomaly and Misuse Detection," in *Proceedings of the 8th USENIX Security Symposium (USENIX Security '99)*, Aug. 1999.

[96] S. Ghavamnia, T. Palit, S. Mishra, and M. Polychronakis, "Temporal System Call Specialization for Attack Surface Reduction," in *Proceedings of the 29th USENIX Security Symposium (USENIX Security'20)*, Aug. 2020.

[97] L. Lei, J. Sun, K. Sun, C. Shenefiel, R. Ma, Y. Wang, and Q. Li, "SPEAKER: Split-Phase Execution of Application Containers," in *Proceedings of the 14th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA'17)*, July 2017.

[98] Z. Gu, B. Saltaformaggio, X. Zhang, and D. Xu, "FACE-CHANGE: Application-Driven Dynamic Kernel View Switching in a Virtual Machine," in *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'14)*, June 2014.

[99] T. Hromatka, "Using a cBPF Binary Tree in Libseccomp to Improve Performance. In *Linux Plumbers Conference (LPC'18)*."

[100] D. Skarlatos, Q. Chen, J. Chen, T. Xu, and J. Torrellas, "Draco: Architectural and Operating System Support for System Call Security," in *Proceedings of the 53rd IEEE/ACM International Symposium on Microarchitecture (MICRO-53)*, Oct. 2020.

[101] A. Grattafiori, "Understanding and Hardening Linux Containers," NCC Group, Tech. Rep., June 2016.

[102] N. DeMarinis, K. Williams-King, D. Jin, R. Fonseca, and V. P. Kemerlis, "sysfilter: Automated System Call Filtering for Commodity Software," in *Proceedings of the 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID'20)*, Oct. 2020.

[103] S. Zhao, R. Gu, H. Qiu, T. O. Li, Y. Wang, H. Cui, and J. Yang, "OWL: Understanding and Detecting Concurrency Attacks," in *Proceedings of the 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'18)*, June 2018.

[104] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding Kernel Race Bugs through Fuzzing," in *Proceedings of the 2019 IEEE Symposium on Security and Privacy (S&P'19)*, May 2019.

[105] R. N. M. Watson, "Exploiting Concurrency Vulnerabilities in System Call Wrappers," in *Proceedings of the 1st USENIX Workshop on Offensive Technologies (WOOT'07)*, Aug. 2007.

[106] S. Gong, D. Altınbüken, P. Fonseca, and P. Maniatis, "Snowboard: Finding Kernel Concurrency Bugs through Systematic Inter-thread Communication Analysis," in *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP'21)*, Oct. 2019.

[107] M. Xu, S. Kashyap, H. Zhao, and T. Kim, "KRACE: Data Race Fuzzing for Kernel File Systems," in *Proceedings of the 2020 IEEE Symposium on Security and Privacy (S&P'20)*, May 2020.

[108] J. Edge, "Deep argument inspection for seccomp," https://lwn.net/Articles/799557/, Sep. 2019.

[109] J. Edge, "Seccomp and deep argument inspection," https://lwn.net/Articles/822256/, June 2020.

[110] J. Edge, "System call filtering and no_new_privs," https://lwn.net/Articles/475678/, Jan. 2012.

[111] "eBPF – Introduction, Tutorials & Community Resources," https://ebpf.io/.

[112] A. Starovoitov, "rework/optimize internal BPF interpreter's instruction set," https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=bd4cf0ed331a275e9bf5a49e6d0fd55dffc551b8, Mar. 2014.

[113] C. Canella, A. Kogler, L. Giner, D. Gruss, and M. Schwarz, "Domain Page-Table Isolation," in *arXiv:2111.10876*, Nov. 2021.

[114] J. Corbet, "Memory protection keys for the kernel," https://lwn.net/Articles/756233/, July 2020.

[115] S. Gravani, M. Hedayati, J. Criswell, and M. L. Scott, "Fast Intra-kernel Isolation and Security with IskiOS," in *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'21)*, Oct. 2021.

[116] T. Frassetto, P. Jauernig, C. Liebchen, and A.-R. Sadeghi, "IMIX: In-Process Memory Isolation EXtension," in *Proceedings of the 27th USENIX Security Symposium (USENIX Security '18)*, Aug. 2018.

[117] "Yama ptrace_scope - the linux kernel archives," https://www.kernel.org/doc/Documentation/security/Yama.txt.

[118] J. Corbet, "Sleepable BPF programs," https://lwn.net/Articles/825415/, July 2020.

[119] A. Starovoitov, "Re: seccomp feature development," https://lwn.net/ml/linux-kernel/CAADnVQKRCCHRQrNy=V7ue38skb8nKCczScpph2WFv7U_jsS3KQ@mail.gmail.com/, May 2020.

[120] A. Starovoitov, "Lifetime of BPF objects," https://facebookmicrosites.github.io/bpf/blog/2018/08/31/object-lifetime.html, Aug. 2018.

[121] "Namespaces in operation, part 1: namespaces overview [lwn.net]," https://lwn.net/Articles/531114/.

[122] J. Frazelle, "Security for the Modern Age," *Communications of the ACM*, vol. 62, no. 1, pp. 43–45, Jan. 2019.

[123] S. Kerner, "The future of Docker containers," https://lwn.net/Articles/788282/, May 2019.

[124] D. Walsh, V. Rothberg, and G. Scrivano, "An introduction to crun, a fast and low-memory footprint container runtime," https://www.redhat.com/sysadmin/introduction-crun, Aug. 2020.

[125] "Bug 9071 - busybox - (local) cmdline stack buffer overwrite," https://bugs.busybox.net/show_bug.cgi?id=9071.

[126] J. Corbet, "Constant-action bitmaps for seccomp()," https://lwn.net/Articles/834785/, Oct. 2020.

[127] "ab - Apache HTTP server benchmarking tool," https://httpd.apache.org/docs/2.4/programs/ab.html.

[128] "NoSQL Redis and Memcache traffic generation and benchmarking tool," https://github.com/RedisLabs/memtier_benchmark.

[129] "DNS Measurement, Troubleshooting and Security Auditing Toolset," https://dnsdiag.org/.

[130] J. Edge, "Kernel runtime security instrumentation," https://lwn.net/Articles/798157/, Sep. 2019.

[131] S. Ghavamnia, T. Palit, S. Mishra, and M. Polychronakis, "Confine: Automated System Call Policy Generation for Container Attack Surface Reduction," in *Proceedings of the 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID'20)*, Oct. 2020.

[132] S. Pailoor, X. Wang, H. Shacham, and I. Dillig, "Automated Policy Synthesis for System Call Sandboxing," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020.

[133] C. Canella, M. Werner, D. Gruss, and M. Schwarz, "Automating Seccomp Filter Generation for Linux Applications," in *Proceedings of the 2021 ACM Cloud Computing Security Workshop (CCSW'21)*, Nov. 2021.

[134] Z. Wan, D. Lo, X. Xia, L. Cai, and S. Li, "Mining Sandboxes for Linux Containers," in *Proceedings of 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST'17)*, Tokyo, Japan, Mar. 2017.

[135] V. Rothberg, "Generate SECCOMP Profiles for Containers Using Podman and eBPF," https://podman.io/blogs/2019/10/15/generate-seccomp-profiles.html, Oct. 2019.

[136] D. Mutz, F. Valeur, G. Vigna, and C. Kruegel, "Anomalous System Call Detection," *ACM Transactions on Information and System Security (TISSEC)*, vol. 9, no. 1, pp. 61–93, Feb. 2006.

[137] L. Nelson, J. V. Geffen, E. Torlak, and X. Wang, "Specification and verification in the field: Applying formal methods to BPF just-in-time compilers in the Linux kernel," in *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, Nov. 2020.

[138] Q. Xu, M. D. Wong, T. Wagle, S. Narayana, and A. Sivaraman, "Synthesizing Safe and Efficient Kernel Extensions for Packet Processing," in *Proceedings of the 2021 ACM SIGCOMM Conference (SIGCOMM'21)*, Aug. 2021.

[139] J. V. Geffen, L. Nelson, I. Dillig, X. Wang, and E. Torlak, "Synthesizing JIT Compilers for In-Kernel DSLs," in *Proceedings of the 32nd International Conference on Computer-Aided Verification (CAV'20)*, July 2020.

[140] X. Wang, D. Lazar, N. Zeldovich, A. Chlipala, and Z. Tatlock., "Jitk: A Trustworthy In-Kernel Interpreter Infrastructure," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, Oct. 2014.

[141] E. Gershuni, N. Amit, A. Gurfinkel, N. Narodytska, J. A. Navas, N. Rinetzky, L. Ryzhyk, and M. Sagiv, "Simple and Precise Static Analysis of Untrusted Linux Kernel Extensions," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'19)*, June 2019.

[142] L. Nelson, J. Bornholt, R. Gu, A. Baumann, E. Torlak, and X. Wang, "Scaling Symbolic Evaluation for Automated Verification of Systems Code with Serval," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)*, Oct. 2019.

[143] N. Elhage, "Supporting Linux kernel development in Rust," https://lwn.net/Articles/829858/, Aug. 2020.

[144] J. Li, S. Miller, D. Zhuo, A. Chen, J. Howell, and T. Anderson, "An Incremental Path towards a Safer OS Kernel," in *Proceedings of the 18th Workshop on Hot Topics in Operating Systems (HotOS'21)*, June 2021. [Online]. Available: https://doi.org/10.1145/3458336.3465277

[145] A. Balasubramanian, M. S. Baranowski, A. Burtsev, A. Panda, Z. Rakamarić, and L. Ryzhyk, "System Programming in Rust: Beyond Safety," in *Proceedings of the 16th ACM Workshop on Hot Topics in Operating Systems (HotOS'17)*, May 2017.

[146] "Fprobe - Function entry/exit probe," https://docs.kernel.org/trace/fprobe.html.

[147] X. J. Ren, K. Rodrigues, L. Chen, C. Vega, M. Stumm, and D. Yuan, "An Analysis of Performance Evolution of Linux's Core Operations," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)*, Oct. 2019.

[148] "Kernel level exception handling," https://docs.kernel.org/arch/x86/exception-tables.html.

[149] A. Lutomirski, "Why do kprobes and uprobes singlestep?" https://lore.kernel.org/all/161469874601.49483.11985325887166921076.stgit@devnote2/T/#mbb8fd3431b354681310a12741adfd57fad0e7d95. (Feb. 2021).

[150] "Machine IR (MIR) Format Reference Manual," https://llvm.org/docs/MIRLangRef.html.

[151] S. Tolvanen, "[PATCH v6 01/18] add support for Clang CFI," https://lore.kernel.org/all/20210408182843.1754385-2-samitolvanen@google.com/. (Apr. 2021).

[152] J. Corbet, "Avoiding retpolines with static calls," https://lwn.net/Articles/815908/. (Mar. 2020).

[153] "ORC unwinder," https://docs.kernel.org/arch/x86/orc-unwinder.html.

[154] J. Corbet, "Indirect branch tracking for Intel CPUs," https://lwn.net/Articles/889475/. (Mar. 2022).

[155] "A Technical Look at Intel's Control-flow Enforcement Technology," https://www.intel.com/content/www/us/en/developer/articles/technical/technical-look-control-flow-enforcement-technology.html.

[156] K. Boos, N. Liyanage, R. Ijaz, and L. Zhong, "Theseus: an Experiment in Operating System Structure and State Management," in *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, Nov. 2020.

[157] V. Narayanan, T. Huang, D. Detweiler, D. Appel, Z. Li, G. Zellweger, and A. Burtsev, "RedLeaf: Isolation and Communication in a Safe Operating System," in *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, Nov. 2020.

[158] D. Jin, A. J. Gaidis, and V. P. Kemerlis, "BeeBox: Hardening BPF against Transient Execution Attacks," in *Proceedings of the 33rd USENIX Security Symposium (USENIX Security'24)*, Aug. 2024.

[159] S. Y. Lim, T. Prasad, X. Han, and T. Pasquier, "SafeBPF: Hardware-assisted Defense-in-depth for eBPF Kernel Extensions," *arXiv:2409.07508*, Sep. 2024.

[160] J. Corbet, "Reconsidering unprivileged BPF," https://lwn.net/Articles/796328/. (Aug. 2019).

[161] J. Edge, "BPF and security," https://lwn.net/Articles/946389/. (Oct. 2023).

[162] P. Gupta, "bpf: Disallow unprivileged bpf by default," https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=8a03e56b253e9691c90bc52ca199323d71b96204. (Oct. 2021).

[163] "Cilium - Cloud Native, eBPF-based Networking, Observability, and Security," https://cilium.io/.

[164] "Aya-rs," https://aya-rs.dev/.

[165] "Katran - A high performance layer 4 load balancer," https://github.com/facebookincubator/katran.

[166] A. Starovoitov, "bpf: verifier (add docs)," https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=51580e798cb61b0fc63fa3aa6c5c975375aa0550. (Sept. 2014).

[167] P. Chaignon, "bpf: Avoid 32bit assignment of packet pointer," https://github.com/cilium/cilium/commit/847014aa62f94e5a53178670cad1eacea455b227. (May 2023).

[168] D. Borkmann, "bpf: fix verifier's ctx port access in post bind hooks," https://github.com/cilium/cilium/commit/394e72478a8d120dab0bff2c41db77695877ce57. (Mar. 2023).

[169] J. Rajahalme, "datapath: Use inline function to keep policy and l4policy checks separate," https://github.com/cilium/cilium/commit/142c0f7128c7fac22eb18b2c21a56433f19a5ef8. (Apr. 2023).

[170] D. Borkmann, "bpf: Fix verifier issue in fib_redirect," https://github.com/cilium/cilium/commit/efb5d6509fea263bd6d36998f8e524d9942b8a79. (Mar. 2023).

[171] S. Bhat and H. Shacham, "Formal Verification of the Linux Kernel eBPF Verifier Range Analysis," https://sanjit-bhat.github.io/assets/pdf/ebpf-verifier-range-analysis22.pdf. (May 2022).

[172] L. Nelson, X. Wang, and E. Torlak, "A proof-carrying approach to building correct and flexible BPF verifiers," in *Linux Plumbers Conference*, Sep. 2021.

[173] S. Y. Lim, X. Han, and T. Pasquier, "Unleashing Unprivileged eBPF Potential with Dynamic Sandboxing," in *Proceedings of the 1st ACM SIGCOMM 2023 Workshop on eBPF and Kernel Extensions (eBPF'23)*, Sep. 2023.

[174] T. Graf, "bpf: Workaround for verifier bug in proxy hairpin code," https://github.com/cilium/cilium/commit/e38a92115620125b19c8761f35f6709e71c34511. (May 2019).

[175] T. Duberstein, "bpf: Remove builtin global functions," https://github.com/aya-rs/aya/pull/698. (July 2023).

[176] H. Vishwanathan, M. Shachnai, P. Chaignon, S. Nagarakatte, and S. Narayana, "Agni: Fast Formal Verification of the Verifier's Range Analysis. In *Linux Plumbers Conference (LPC'24)*," https://lpc.events/event/18/contributions/1937/. (Sept. 2024).

[177] M. Fahndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. Larus, and S. Levi, "Language Support for Fast and Reliable Message-based Communication in Singularity OS," in *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys'06)*, Apr. 2006.

[178] J. Corbet, "A first look at Rust in the 6.1 kernel," https://lwn.net/Articles/910762/. (Oct. 2022).

[179] A. Levy, B. Campbell, B. Ghena, D. B. Giffin, P. Pannuto, P. Dutta, and P. Levis, "Multiprogramming a 64kB Computer Safely and Efficiently," in *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP'17)*, Oct. 2017.

[180] T. H. Kim, D. Rudo, K. Zhao, Z. N. Zhao, and D. Skarlatos, "Perspective: A Principled Framework for Pliable and Secure Speculation in Operating Systems," in *Proceedings of the ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA'24)*, June 2024.

[181] A. Murray, "Unprivileged eBPF disabled by default for Ubuntu 20.04 LTS, 18.04 LTS, 16.04 ESM," https://discourse.ubuntu.com/t/unprivileged-ebpf-disabled-by-default-for-ubuntu-20-04-lts-18-04-lts-16-04-esm/27047. (Mar. 2022).

[182] "Security Hardening: Use of eBPF by unprivileged users has been disabled by default," https://www.suse.com/support/kb/doc/?id=000020545.

[183] S. R. Somaraju, S. Chintamaneni, and D. Williams, "Overflowing the kernel stack with BPF. In *Linux Plumbers Conference (LPC'23)*," https://lpc.events/event/17/contributions/1595/. (Nov. 2023).

[184] R. Sahu and D. Williams, "When BPF programs need to die: exploring the design space for early BPF termination. In *Linux Plumbers Conference (LPC'23)*," https://lpc.events/event/17/contributions/1610/. (Nov. 2023).

[185] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, "RustBelt: securing the foundations of the Rust programming language," *Proceedings of the 45th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'18)*, Jan. 2018.

[186] R. Jung, H.-H. Dang, J. Kang, and D. Dreyer, "Stacked borrows: an aliasing model for Rust," *Proceedings of the 47th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'20)*, Jan. 2020.

[187] A. Lattuada, T. Hance, C. Cho, M. Brun, I. Subasinghe, Y. Zhou, J. Howell, B. Parno, and C. Hawblitzel, "Verus: Verifying Rust Programs using Linear Ghost Types," *Proceedings of the 2023 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'23)*, Apr. 2023.

[188] A. Lattuada, T. Hance, J. Bosamiya, M. Brun, C. Cho, H. LeBlanc, P. Srinivasan, R. Achermann, T. Chajed, C. Hawblitzel, J. Howell, J. R. Lorch, O. Padon, and B. Parno, "Verus: A Practical Foundation for Systems Verification," in *Proceedings of the 30th ACM Symposium on Operating Systems Principles (SOSP'24)*, Sep. 2024.

[189] "Rust Verification Tools," https://project-oak.github.io/rust-verification-tools/.

[190] "Fuzzing," https://rustc-dev-guide.rust-lang.org/fuzzing.html.

[191] "The rustc book," https://doc.rust-lang.org/stable/rustc/.

[192] "What is RCU? – "Read, Copy, Update"," https://docs.kernel.org/RCU/whatisRCU.html.

[193] "std - Rust," https://doc.rust-lang.org/std/index.html.

[194] "no_std - The Embedded Rust Book," https://docs.rust-embedded.org/book/intro/no-std.html.

[195] "alloc - Rust," https://doc.rust-lang.org/alloc/index.html.

[196] "Lints - The rustc book," https://doc.rust-lang.org/rustc/lints/index.html.

[197] "Clippy Lints," https://rust-lang.github.io/rust-clippy/master/index.html.

[198] "Codegen Options - The rustc book," https://doc.rust-lang.org/rustc/codegen-options/index.html.

[199] "Generic parameters," https://doc.rust-lang.org/reference/items/generics.html.

[200] "Monomorphization," https://rustc-dev-guide.rust-lang.org/backend/monomorph.html.

[201] D. Vernet, "More flexible memory access for BPF programs," https://lwn.net/Articles/910873/. (Oct. 2022).

[202] "auto_traits," https://doc.rust-lang.org/beta/unstable-book/language-features/auto-traits.html.

[203] "negative_impls," https://doc.rust-lang.org/beta/unstable-book/language-features/negative-impls.html.

[204] "Trait objects," https://doc.rust-lang.org/reference/types/trait-object.html.

[205] "Bounds - Rust By Example," https://doc.rust-lang.org/rust-by-example/generics/bounds.html.

[206] "RAII - Rust By Example," https://doc.rust-lang.org/rust-by-example/scope/raii.html.

[207] "Drop in core::ops - Rust," https://doc.rust-lang.org/core/ops/trait.Drop.html.

[208] "Itanium C++ ABI: Exception Handling ($Revision: 1.22 $)," http://itanium-cxx-abi.github.io/cxx-abi/abi-eh.html.

[209] S. Chintamaneni, S. R. Somaraju, and D. Williams, "Unsafe kernel extension composition via BPF program nesting," in *Proceedings of the 2nd ACM SIGCOMM 2024 Workshop on eBPF and Kernel Extensions (eBPF'24)*, Aug. 2024.

[210] T. Gleixner and I. Molnar, "hrtimers - subsystem for high-resolution kernel timers," https://docs.kernel.org/timers/hrtimers.html.

[211] A. Chaiken, "IRQs: the Hard, the Soft, the Threaded and the Preemptible. In *Embedded Linux Conference Europe (ELCE'16)*," https://events.static.linuxfound.org/sites/events/files/slides/Chaiken_ELCE2016.pdf. (Oct. 2016).

[212] "rust-bindgen," https://github.com/rust-lang/rust-bindgen.

[213] "The LLVM Compiler Infrastructure," https://llvm.org/.

[214] J. Wagner, V. Kuznetsov, and G. Candea, "-OVERIFY: Optimizing Programs for Fast Verification," in *Proceedings of the 14th USENIX Workshop on Hot Topics in Operating Systems (HotOS'13)*, May 2013.

[215] Y. Zhang, P. Li, Y. Ding, L. Wang, N. Meng, and D. Williams, "Broadly Enabling KLEE to Effortlessly Find Unrecoverable Errors," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP'24)*, Apr. 2024.

[216] "Kani Rust Verifier," https://github.com/model-checking/kani.

[217] "The Coq Proof Assistant," https://coq.inria.fr/.

[218] "The Dafny Programming and Verification Language," https://dafny.org/.

[219] K. K. Dwivedi, "bpf: Introduce bpf_obj_new," https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=958cf2e273f0929c66169e0788031310e8118722. (Nov. 2022).

[220] J. Corbet, "Calling kernel functions from BPF," https://lwn.net/Articles/856005/. (May 2021).

[221] J. Corbet, "A BPF-specific memory allocator," https://lwn.net/Articles/899274/. (June 2022).

[222] N. Provos, "Improving Host Security with System Call Policies," in *Proceedings of the 12th USENIX Security Symposium (USENIX Security '03)*, Aug. 2003.

[223] K. Jain and R. Sekar, "User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement," in *Proceedings of the 2000 Network and Distributed System Security Symposium (NDSS'00)*, Feb. 2000.

[224] A. Acharya and M. Raje, "MAPbox: Using Parameterized Behavior Classes to Confine Untrusted Applications," in *Proceedings of the 9th USENIX Security Symposium (USENIX Security '00)*, Aug. 2000.

[225] T. Kim and N. Zeldovich, "Practical and Effective Sandboxing for Non-root Users," in *Proceedings of the 2013 USENIX Conference on Annual Technical Conference (USENIX ATC'13)*, June 2013.

[226] A. Alexandrov, P. Kmiec, and K. Schauser, "Consh: Confined Execution Environment for Internet Computations," The University of California, Santa Barbara, Tech. Rep., 1999.

[227] D. S. Peterson, M. Bishop, and R. Pandey, "A Flexible Containment Mechanism for Executing Untrusted Code," in *Proceedings of the 11th USENIX Security Symposium (USENIX Security '02)*, Aug. 2002.

[228] T. Fraser, L. Badger, and M. Feldman, "Hardening COTS Software with Generic Software Wrappers," in *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, May 1999.

[229] A. Dan, A. Mohindra, R. Ramaswami, and D. Sitara, "ChakraVyuha (CV): A Sandbox Operating System Environment for Controlled Execution of Alien Code," IBM Research Division, T.J. Watson Research Center, Tech. Rep. RC 20742 (2/20/97), Feb. 1997.

[230] D. P. Ghormley, D. Petrou, S. H. Rodrigues, and T. E. Anderson, "SLIC: An Extensibility System for Commodity Operating Systems," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference (USENIX ATC'98)*, June 1998.

[231] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch, "Leveraging Legacy Code to Deploy Desktop Applications on the Web," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, Dec. 2008.

[232] K. Cook, "Re: [PATCH net-next 0/3] eBPF Seccomp filters," https://lore.kernel.org/netdev/CAGXu5jLiYh0rSRuJ_-2xLB03Wod5G07njpoESR4SnmsmiUnsEw@mail.gmail.com/ (Feb. 2018).

[233] A. Lutomirski, "Re: [rfc patch bpf-next seccomp 00/12] ebpf seccomp filters," https://lore.kernel.org/bpf/b3a1684b-86e4-74c4-184b-7700613aa838@kernel.org/. (May. 2021).

[234] K. Cook, "Re: [rfc patch bpf-next seccomp 00/12] ebpf seccomp filters," https://lore.kernel.org/bpf/202106011244.76762C210@keescook/. (Jun. 2021).

[235] B. Rhoden, "eBPF Kernel Scheduling with Ghost. In *Linux Plumbers Conference (LPC'22)*," https://lpc.events/event/16/contributions/1365/. (Sept. 2022).

[236] D. Skarlatos and K. Zhao, "Towards Programmable Memory Management with eBPF. In *Linux Plumbers Conference (LPC'24)*," https://lpc.events/event/18/contributions/1932/. (Sept. 2024).

[237] N. Amit and M. Wei, "The Design and Implementation of Hyperupcalls," in *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC'18)*, July 2018.

[238] V. Prasad, W. Cohen, F. Eigler, M. Hunt, J. Keniston, and B. Chen, “Locating System Problems Using Dynamic Instrumentation,” in *Proceedings of the 2005 Ottawa Linux Symposium*, 2005.

[239] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, “Dynamic Instrumentation of Production Systems,” in *Proceedings of the 2004 USENIX Annual Technical Conference (USENIX ATC'04)*, June 2004.

[240] “ptrace(2) — linux manual page,” https://man7.org/linux/man-pages/man2/ptrace.2.html.

[241] N. A. Quynh and K. Suzaki, “Xenprobes, A Lightweight User-space Probing Framework for Xen Virtual Machine,” in *Proceedings of the 2007 USENIX Annual Technical Conference (USENIX ATC'07)*, June 2007.

[242] B. D. Payne, M. Carbone, M. Sharif, and W. Lee, “Lares: An Architecture for Secure Active Monitoring Using Virtualization,” in *Proceedings of the 2008 IEEE Symposium on Security and Privacy (S&P'08)*, May 2008.

[243] Z. J. Estrada, C. Pham, F. Deng, L. Yan, Z. Kalbarczyk, and R. K. Iyer, “Dynamic VM Dependability Monitoring Using Hypervisor Probes,” in *Proceedings of the 11th European Dependable Computing Conference (EDCC'15)*, Sep. 2015.

[244] P. F. Klemperer, H. Y. Jeon, B. D. Payne, and J. C. Hoe, “High-Performance Memory Snapshotting for Real-Time, Consistent, Hypervisor-Based Monitors,” *IEEE Transactions on Dependable and Secure Computing (TDSC)*, vol. 17, no. 3, pp. 518–535, Feb. 2018.

[245] J. Reeves, A. Ramaswamy, M. Locasto, S. Bratus, and S. Smith, “Intrusion detection for resource-constrained embedded control systems in the power grid,” *International Journal of Critical Infrastructure Protection (IJCIP)*, vol. 5, no. 2, pp. 74–83, July 2012.

[246] “ftrace - Function Tracer,” https://docs.kernel.org/trace/ftrace.html.

[247] D. Bruening, “Efficient, Transparent, and Comprehensive Runtime Code Manipulation,” Ph.D. dissertation, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2004.

[248] A. R. Bernat and B. P. Miller, “Anywhere, Any-Time Binary Instrumentation,” in *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'11)*, Sep. 2011.

[249] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, “Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation,” in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*, June 2005.

[250] M.-K. Yoon, S. Mohan, J. Choi, J.-E. Kim, and L. Sha, "SecureCore: A multicore-based intrusion detection architecture for real-time embedded systems," in *Proceedings of the 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS'13)*, Apr. 2013.

[251] B. Niu and G. Tan, "Per-Input Control-Flow Integrity," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS'15)*, Oct. 2015.

[252] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical Context-Sensitive CFI," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS'15)*, Oct. 2015.

[253] R. Ding, C. Qian, C. Song, B. Harris, T. Kim, and W. Lee, "Efficient Protection of Path-Sensitive Control Security," in *Proceedings of the 26th USENIX Security Symposium (USENIX Security'17)*, Aug. 2017.

[254] Y. Liu, P. Shi, X. Wang, H. Chen, B. Zang, and H. Guan, "Transparent and Efficient CFI Enforcement with Intel Processor Trace," in *Proceedings of the 23rd IEEE Symposium on High Performance Computer Architecture (HPCA'17)*, Feb. 2017.

[255] H. Hu, C. Qian, C. Yagemann, S. P. H. Chung, W. R. Harris, T. Kim, and W. Lee, "Enforcing Unique Code Target Property for Control-Flow Integrity," in *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS'18)*, Oct. 2018.

[256] M. R. Khandaker, W. Liu, A. Naser, Z. Wang, and J. Yang, "Origin-sensitive Control Flow Integrity," in *Proceedings of the 28th USENIX Security Symposium (USENIX Security'19)*, Aug. 2019.

[257] S. Yoo, J. Park, S. Kim, Y. Kim, and T. Kim, "In-Kernel Control-Flow Integrity on Commodity OSes using ARM Pointer Authentication," in *Proceedings of the 31st USENIX Security Symposium (USENIX Security'22)*, Aug. 2022.

[258] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of Control: Overcoming Control-Flow Integrity," in *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P'14)*, May 2014.

[259] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection," in *Proceedings of the 23rd USENIX Security Symposium (USENIX Security'14)*, Aug. 2014.

[260] H.-W. Hung and A. Amiri Sani, "BRF: Fuzzing the eBPF Runtime," in *Proceedings of the 2024 ACM International Conference on the Foundations of Software Engineering (FSE'24)*, July 2024.

[261] H. Sun, Y. Xu, J. Liu, Y. Shen, N. Guan, and Y. Jiang, "Finding Correctness Bugs in eBPF Verifier with Structured and Sanitized Program," in *Proceedings of the 19th ACM European Conference on Computer Systems (EuroSys'24)*, Apr. 2024.

[262] H. Sun and Z. Su, "Validating the eBPF Verifier via State Embedding," in *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI'24)*, July 2024.

[263] M. H. N. Mohamed, X. Wang, and B. Ravindran, "Understanding the Security of Linux eBPF Subsystem," in *Proceedings of the 14th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys'23)*, Aug. 2023.

[264] H. Vishwanathan, M. Shachnai, S. Narayana, and S. Nagarakatte, "Verifying the Verifier: eBPF Range Analysis Verification," in *Proceedings of the 35th International Conference on Computer Aided Verification (CAV'23)*, July 2023.

[265] P. Zhang, C. Wu, X. Meng, Y. Zhang, M. Peng, S. Zhang, B. Hu, M. Xie, Y. Lai, Y. Kang, and Z. Wang, "HIVE: A Hardware-assisted Isolated Execution Environment for eBPF on AArch64," in *Proceedings of the 33rd USENIX Security Symposium (USENIX Security'24)*, Aug. 2024.

[266] H. Lu, S. Wang, Y. Wu, W. He, and F. Zhang, "MOAT: Towards Safe BPF Kernel Extension," in *Proceedings of the 33rd USENIX Security Symposium (USENIX Security'24)*, Aug. 2024.

[267] G. Hunt, J. Larus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill, "An Overview of the Singularity Project," Microsoft Research. (Oct. 2005), Tech. Rep. MSR-TR-2005-135.

[268] A. Burtsev, V. Narayanan, Y. Huang, K. Huang, G. Tan, and T. Jaeger, "Evolving Operating System Kernels Towards Secure Kernel-Driver Interfaces," in *Proceedings of the 19th ACM Workshop on Hot Topics in Operating Systems (HotOS'23)*, June 2023.

[269] G. C. Necula and P. Lee, "Safe Kernel Extensions Without Run-Time Checking," in *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI'96)*, Oct. 1996.

[270] D. Weston, "The journey towards default security. In *BlueHat IL 2023*," https://www.youtube.com/watch?v=8T6ClX-y2AE. (Mar. 2023).