Revealing the Unstable Foundations of eBPF-Based Kernel Extensions

Shawn (Wanxiang) Zhong
University of Wisconsin-Madison

Jing Liu[*]
Microsoft Research

Andrea Arpaci-Dusseau
University of Wisconsin-Madison

Remzi Arpaci-Dusseau
University of Wisconsin-Madison

Abstract

eBPF programs significantly enhance kernel capabilities, but encounter substantial compatibility challenges due to their deep integration with unstable kernel internals. We introduce DepSurf, a tool that identifies dependency mismatches between eBPF programs and kernel images. Our analysis of 25 kernel images spanning 8 years reveals that dependency mismatches are pervasive, stemming from kernel source code evolution, diverse configuration options, and intricate compilation processes. We apply DepSurf to 53 real-world eBPF programs, and find that 83% are impacted by dependency mismatches, underscoring the urgent need for systematic dependency analysis. By identifying these mismatches, DepSurf enables a more robust development and maintenance process for eBPF programs, enhancing their reliability across a wide range of kernels.

ACM Reference Format:
Shawn (Wanxiang) Zhong, Jing Liu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2025. Revealing the Unstable Foundations of eBPF-Based Kernel Extensions. In *Twentieth European Conference on Computer Systems (EuroSys '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 21 pages. https://doi.org/10.1145/3689031.3717497

1 Introduction

Software is not executed in isolation, but rather depends strongly on the environment around it to function correctly. For example, a C program running on Linux will execute as planned only if both the program itself is correct, and also the libraries and system calls it relies on [16].

One new software environment that has been gaining importance is eBPF [74]. eBPF enables developers to create programs that can be run in the kernel, thus extending kernel capabilities in powerful ways; sandboxing technology enables this code to be run safely within the trusted confines of the kernel. Many eBPF applications exist and are widely used in domains such as networking [18, 26, 49], storage [12, 177, 183], security [27, 29, 97, 143], scheduling [1], and monitoring and observability [74, 75, 138].

Unfortunately, the software environment for eBPF programs poses significant challenges to developers in both crafting correct programs and maintaining them across different Linux kernels. eBPF programs are event-driven, triggered by hooks (e.g., kernel functions) that can change across versions. They also peer into kernel data structures and function parameters; these too are often modified over time. Adding to the challenge are varying kernel configurations that alter the presence and definition of kernel functions and structs, as well as the intricacies of the compilation process (such as function inline). All of these dependencies on unstable kernel internals make realizing correct eBPF programs difficult across kernels. For example, the `biotop` program [76], which monitors block I/O activity, stopped working in Linux v5.16 when an innocuous commit inlined two critical functions it relied upon. This issue persisted for two years and required collaboration between eBPF developers and kernel maintainers to finally resolve.

While these dependencies are crucial for eBPF programs, there has not yet been a systematic study of eBPF dependency issues. This paper explores several key questions about these dependencies: What kernel internals do eBPF programs depend on? How unstable are they? More importantly, how do they affect real-world eBPF programs in practice?

In this paper, we introduce DepSurf, a tool designed to identify dependency mismatches between eBPF programs and kernel images. DepSurf operates in two stages. It first analyzes kernel images to extract the *dependency surface*: all kernel constructs (such as functions, structs, and tracepoints) that eBPF programs can use. This extracted data is then processed to create a dataset of potential mismatches that would occur if any of these constructs were used. Second, DepSurf examines an eBPF program to produce its *dependency set*: the set of kernel constructs that the program relies upon for correct operation. By querying the collected dataset against the dependency set, DepSurf automatically generates a report for dependency mismatches. The source code of DepSurf and the dataset are publicly released to

[*] Work partly done while at University of Wisconsin-Madison

EuroSys '25, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1196-1/2025/03
https://doi.org/10.1145/3689031.3717497

facilitate further research[1].

DepSurf serves two objectives: to enhance the robustness of eBPF program development and maintenance, and to comprehensively study eBPF dependency issues.

For developing new eBPF programs, DepSurf enables efficient compatibility checks across a wide range of kernel images, reducing errors and uncovering potential silent failures. For existing eBPF programs, DepSurf allows developers to proactively identify mismatches on new kernels, allowing them to address issues in advance.

Using DepSurf, we conduct the first comprehensive analysis of eBPF dependency issues across 25 kernel images, spanning 17 kernel versions over 8 years, 5 architectures, 5 configuration flavors, and 14 compiler versions. We identify three contributors to mismatches: kernel source changes, configuration variations, and intricacies of the compilation process. Our findings reveal constant changes to the dependency surface, with 6% of functions changing signatures every 2 years, resulting in silent errors for dependent eBPF programs. Even tracepoints, which are presumed to be stable, show 5% removal and 16% changes. Configuration options strongly impact construct presence, with about a quarter of the constructs absent across configurations. Compiler optimizations cause further mismatches with 36% of functions selectively inlined (i.e., inlined at some call sites but not others), leading to incomplete results, and 16% transformed with different signatures. These findings demonstrate the unstable nature of the eBPF dependency surface.

We apply DepSurf to analyze 53 real-world eBPF programs. Our analysis reveals that eBPF programs have diverse dependency sets: 79% rely on structs or fields, 47% depend on functions, and another 47% utilize tracepoints. Notably, our findings indicate that dependency mismatches are pervasive, affecting the vast majority (83%) of the analyzed programs. These mismatches make it challenging for developers to maintain compatibility across kernels, underscoring the need for automated tools like DepSurf for dependency analysis.

Contributions. We make the following contributions:

- We formalize the concept of a dependency surface and present the first systematic study of dependency issues for eBPF-based kernel extensions.
- We design DepSurf, an automated tool that diagnoses dependency mismatches for eBPF programs, allowing developers to build and maintain robust eBPF programs.
- We develop a novel approach to analyze compiled kernel images, enabling new opportunities for kernel research.
- We release a dataset of dependency surfaces and mismatches to facilitate research on dependency issues.
- We apply DepSurf to analyze real-world eBPF programs, offering insights into dependency mismatches in practice.

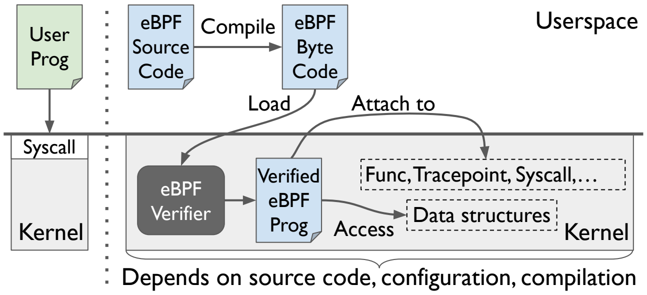


Figure 1. Overview of eBPF-based kernel extensions.

2 Background and Motivation

2.1 Prevalence of eBPF-based Kernel Extensions

Kernel extensions have been a recurring topic in operating system research [10, 46, 48, 68, 78, 87, 115, 121, 144]; however, only with the recent rise of eBPF are such extensions becoming widely adopted, enabling production-ready user programs to customize kernel behaviors without modifying the kernel [54, 75]. As Linus Torvalds said, eBPF "has actually been really useful, and the real power of it is how it allows people to do specialized code that isn't enabled until asked for" [179]; Brendan Gregg called eBPF "superpowers for Linux" [73]. Indeed, eBPF programs are used for a variety of functionality, including observability, security, and networking.

Observability. eBPF programs can extract performance metrics from the kernel by running custom code to gather and process data at points of interest. Projects such as BCC [75] and bpftrace [138] use eBPF to provide a high-level tracing interface, enabling users to write simple scripts to troubleshoot performance issues and debug kernel internals in real deployments [70, 101]. eBPF is used for observability in companies like Google [147], Microsoft [42], and Cloudflare [28].

Security. Security tools such as Falco [156], Tracee [143], Tetragon [27], and Pixie [128] use eBPF to detect malicious activities. Security policies can also be enforced, as in lockc [106], bpflock [105], KubeArmor [97], eBPFGuard [43, 44], and systemd [64, 94, 157]. eBPF is widely used for security in production [28, 140, 147].

Networking. eBPF programs can customize network functions, such as firewalls, load balancers, and traffic shaping, with low overhead. Networking projects that use eBPF include Cilium [26], Calico [18], and Katran [49]; eBPF network tools are also used throughout industry [28, 50, 51, 160, 176].

2.2 eBPF-Based Kernel Extensions

Figure 1 shows how eBPF-based kernel extensions work and the changes they bring to the kernel ecosystem.

eBPF programs are typically written in C and subsequently compiled into eBPF bytecode. These programs function in an event-driven manner: they are attached to hooks in the kernel and executed when the hooks are triggered. Running an eBPF program involves two phases: loading and attaching. First, a user program loads the eBPF bytecode into the kernel,

[1] The code and dataset are available at https://github.com/ShawnZhong/DepSurf and https://github.com/ShawnZhong/DepSurf-dataset.

```
1 SEC("kprobe/do_unlinkat")    // attach to func do_unlinkat
2 int trace_unlink(struct pt_regs *ctx) {
3   const char *str;
4   struct filename *f = (void *)ctx->si;    // read 2nd arg
5   bpf_probe_read(&str, sizeof(str), &f->name);    // deref
6   bpf_printk("Unlinking %s", str);               // print
7 }
```

Listing 1. An eBPF program that prints unlinked file names.

where a verifier checks the bytecode to ensure it is safe to run in the kernel. Second, the eBPF program is attached to specific hooks within the kernel, and thus becomes ready to execute.

For example, the eBPF program in Listing 1 attaches to the kernel function `do_unlinkat` and prints the name of the file being unlinked. The function `trace_unlink` takes a single `ctx` argument of type `struct pt_regs *` (line 2), which contains the register values at the time of the kernel function call. This allows eBPF programs to access the arguments passed to the kernel function (line 4). Interaction with the kernel is through roughly 200 eBPF helper functions and 100 kfuncs implemented in the kernel [55, 125]. In lines 5 and 6, `bpf_probe_read` reads kernel memory, and `bpf_printk` prints a message to the kernel log.

eBPF programs can attach to various types of hooks in the kernel [58]. The example above demonstrates a *kprobe*, which enables dynamic instrumentation of over 35,000 kernel functions [60]. Another common type is *tracepoints*, which are static markers in the kernel code by developers [17, 62]. System calls also serve as attachment points for eBPF programs. There are other subsystems that define other types of hooks, such as uprobe [63] for user-space tracing, XDP [135] and tc [126] for network, LSM hooks [57, 61] for security, and `sched-ext` [1] for task scheduling.

2.3 Dependency Surface

We define the *dependency surface* as all constructs (such as functions, structs, and tracepoints) within a kernel image that eBPF programs can depend on. As shown in Figure 1, and unlike narrow APIs available in other programming environments (e.g., system calls), a dependency surface is incredibly broad, including tens of thousands of kernel functions and data structures.

We refer to the set of dependencies that an eBPF program relies on as a *dependency set*. For instance, in Listing 1, the program depends on the kernel function `do_unlinkat` (with a second argument of type `struct filename`), and two structs: `struct filename` (with a field `name` of type `char`) and `struct pt_regs` (with a field `si`).

A *dependency mismatch* occurs when a dependency of an eBPF program is either absent from the kernel's dependency surface or differs from the program's expectations. For instance, if the kernel function `do_unlinkat` directly accepted `char *` instead of `struct filename *` as its second argument (as before Linux v4.15), the program will (silently) produce incorrect results.

2.4 eBPF Portability

Unstable kernel internals create compatibility issues for eBPF programs, undermining their portability. In this section, we discuss the expectations of eBPF portability, the challenges posed by unstable dependency surfaces, and the impact of dependency issues.

Portability Expectations and Practices. The growing adoption of eBPF has created strong expectations for portability across different kernels.

Early eBPF developers relied on conditional compilation directives (e.g., `#if LINUX_VERSION_CODE < ...`) for source code portability. However, such version checks are error-prone and unreliable [30, 114, 149] because of backported kernel patches and other custom modifications [52, 93, 104].

eBPF developers have now widely adopted CO-RE (Compile Once, Run Everywhere) to achieve binary portability [72, 118, 119]. CO-RE provides the mechanism to "produce a single executable binary that you can run on multiple kernel versions and configurations" [53]. CO-RE uses relocation during loading to avoid recompiling eBPF programs [56]. We explain the technical details of relocation in Section 7. Relocation, however, cannot handle cases where recompilation on the target kernel would have failed (e.g., when a field does not exist). For this, CO-RE provides query interfaces (e.g., `bpf_core_field_exists`) that allow developers to inspect type information from the kernel and implement specific logic based on the results.

With CO-RE, eBPF developers often make broad compatibility claims for their programs, such as supporting "Linux 4.1 and above" [75], working with "most common Linux distributions and kernels" [143], and even asserting their program "doesn't require a specific kernel version" [156]. Unfortunately, CO-RE only provides the mechanisms for portability - developers still need to manually identify kernel differences, add appropriate checks, and implement necessary logic in their code. This remains a significant challenge [45, 65], requiring deep understanding of unstable kernel internals.

Reality with Unstable Dependency Surfaces. We now examine the reality of eBPF portability with unstable dependency surfaces. We focus on three critical kernel constructs frequently used by eBPF programs: kernel functions, structs, and tracepoints.

Kernel functions and structs are unstable, but widely used. eBPF programs can attach to tens of thousands of internal kernel functions and read thousands of kernel data structures, which are not guaranteed to be stable. Kernel source code is modified continuously with functions and data structures evolving regularly. Kernel configurations introduce further variability, changing their definitions to tailor to specific options. Even decisions in the compilation process, such as function inline, can impact the presence of functions. Despite the unstable nature, these functions and structures are still widely used in eBPF programs to achieve "superpowers".

```
- void blk_account_io_start() { if (...) /* do work */ }
+ static inline void blk_account_io_start()
+     { if (...) __blk_account_io_start(); }
+ void __blk_account_io_start() { /* do work */ }
```

Listing 2. An excerpt of kernel commit be6bfe3 that broke biotop.

Tracepoints should be stable, but are not. Due to the unstable nature of kernel functions and structs, people have been attempting to use tracepoints as a stable alternative. However, the stability of tracepoints remains a contentious issue [33]. As described in Brendan Gregg's book, *BPF Performance Tools*, "tracepoints provide a stable API," with a footnote stating "I have seen tracepoints change" [74]. Linus Torvalds stands with the #1 rule of kernel development, "don't break userspace" [162–164], elaborating that if a tracepoint change breaks userspace programs, the change will be reverted. However, the rule is not properly enforced [178]. As we will show later, the reality is that tracepoint changes happen surprisingly often, and such changes can break dependent eBPF programs.

Impacts. Dependency issues have far-reaching impacts for users, eBPF developers, and kernel developers.

Users. For users, dependency mismatches undermine confidence in the reliability and correctness of eBPF programs. At best, users encounter explicit errors during compilation or loading. More concerning are silent failures, where programs malfunction without any apparent indication. They can lead to undetected errors or security vulnerabilities that can remain unnoticed for a long time [80, 81, 95, 129]. Dependency mismatches also challenge the long-standing promise of not breaking userspace. As a result, users may become hesitant to upgrade their kernels, fearing potential disruptions to their eBPF-based applications.

eBPF developers. Dependency mismatches greatly complicate development and maintenance for eBPF developers [45, 65]. They cannot be certain if the unstable kernel constructs they rely on are available for their users, and they need to constantly update their programs to maintain compatibility.

Kernel developers. Kernel developers have traditionally been cautious about exposing new interfaces. Some are even reluctant to add new tracepoints due to concerns about maintaining stability [33]. As eBPF programs gain popularity, kernel developers could one day find themselves unable to modify an internal interface due to widespread usage. This phenomenon, known as Hyrum's law [174], has manifested in the Linux kernel [127, 168], as well as numerous other software systems [4, 9, 41, 131].

2.5 Case Study: A Two-Year Journey to Fix biotop

We present a case study of dependency mismatches in an eBPF program biotop [76] to illustrate challenges in diagnosing and fixing dependency mismatches. biotop is designed to monitor block-level I/O latency. It attaches to a pair of kernel functions at the start and end of each I/O operation to record the time difference.

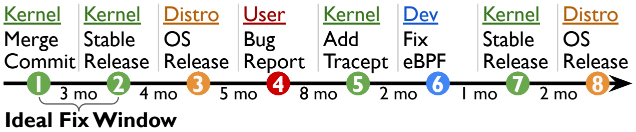


Figure 2. The timeline of fixing a dependency mismatch in biotop.

Figure 2 illustrates the timeline of the issue: ❶ During the development of Linux v5.16, a commit be6bfe3 (see Listing 2) changed a pair of functions (`blk_account_io_{start, done}`) to `static inline`. Both are used by biotop. This change made them inaccessible for attachment and broke the program. ❷ 3 months later, v5.16 was released. Ideally, a fix should have been made before the kernel release. ❸ After 4 months, a major Linux distribution, Fedora 36, was released with a new kernel. ❹ 5 months later, a user reported a bug [132], citing a "failed to attach" error when running biotop on Fedora 36. Developers first attempted to attach to `__blk_account_io_{start, done}` instead, as these were called by the original functions. However, this proved ineffective because the compiler happened to inline the `start` function, despite it not being explicitly marked as `inline`. ❺ Developers proposed adding two tracepoints (`block_io_{start, done}`), which were merged in Linux v6.5 (5a80bd0), 8 months after the bug report. ❻ 2 months later, biotop was updated to use the new tracepoints [112]. ❼ 1 month later, the new kernel with the tracepoints was released. ❽ After 2 months, Fedora 39 was released with the new kernel.

In total, 2 years passed from the merge of the problematic commit to the final fix. Despite the lengthy effort, the new tracepoints are only available for newer kernel versions, and biotop remains broken from v5.17 to v6.4. These delays could have been significantly reduced if developers had been able to detect the dependency mismatch early, allowing them to address it before the stable kernel release (e.g., by implementing tracepoints alongside the initial commit).

However, no tool or automated process currently exists for such early detection. We contacted a core BCC developer, who responded that "any automation will be good going forward." We believe there is a pressing need for automated analysis to better understand dependency mismatches, and facilitate early detection and resolution. We revisit this example in Section 3.3 to show how we help diagnose and fix this case.

3 DepSurf Overview

We develop an automated tool called DepSurf with two objectives: to conduct comprehensive analyses of eBPF dependency mismatches, and to provide developers with diagnostics for early detection and resolution of dependency mismatches. These diagnostics help developers build robust eBPF programs and efficiently maintain existing ones. In this section, we outline the design of DepSurf, detail our methodology for the study, demonstrate usage scenarios, and discuss the implementation details.

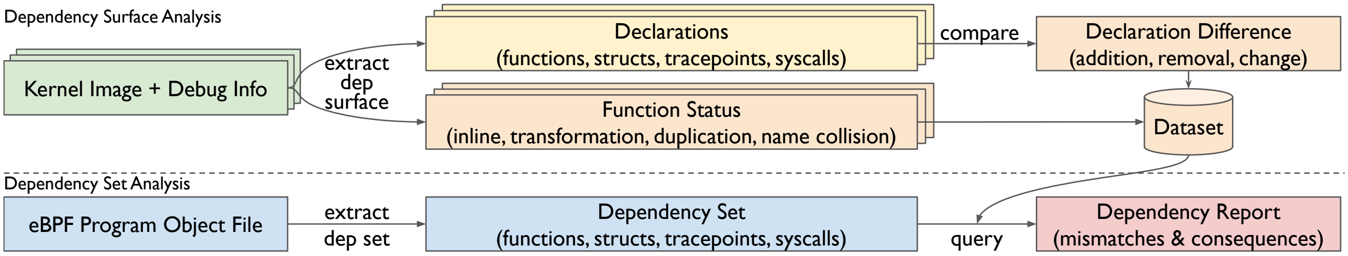

Dependency Surface Analysis
Kernel Image + Debug Info
extract dep surface
Declarations (functions, structs, tracepoints, syscalls)
Function Status (inline, transformation, duplication, name collision)
compare
Declaration Difference (addition, removal, change)
Dataset
Dependency Set Analysis
eBPF Program Object File
extract dep set
Dependency Set (functions, structs, tracepoints, syscalls)
query
Dependency Report (mismatches & consequences)

Figure 3. Design of DepSurf.

3.1 Design

We designed DepSurf to focus on the runtime environment of eBPF programs: the kernel images. As illustrated in Figure 3, DepSurf operates in two stages: analysis of the dependency surfaces exposed by kernel images, and analysis of the dependency set used by eBPF programs.

Dependency Surface Analysis. DepSurf processes a set of kernel images and the associated debug information to analyze potential mismatches within their dependency surfaces. For each image, it extracts two types of data: the declarations of kernel constructs (i.e., functions, structs, tracepoints, and system calls) and function status (e.g., inline or transformation by compiler). DepSurf then compares the declarations across kernel images to detect any constructs that have been added, removed, or changed. For changed constructs, DepSurf records the specific reasons (e.g., function parameter added). This forms a dataset of dependency mismatches that could arise if the constructs were used by an eBPF program.

Dependency Set Analysis. Using the dataset, DepSurf performs dependency set analysis on an eBPF program to assess its dependency mismatches across the set of kernel images. DepSurf extracts the program's dependency sets from the eBPF object file, and looks up the dataset. It then generates a report of mismatches, consequences, and implications on each kernel image.

3.2 Analysis Methodology

Our study analyzes dependency mismatches by examining a diverse range of kernel images. We identify three contributors to the dependency surfaces: the kernel source code, configuration, and compilation process.

Kernel Source Code. Kernel source code is the most obvious source of dependencies. For an eBPF program to function properly, each of its dependencies must be present in the kernel source code. However, the kernel is a large and complex system with new features, refactoring, and bug fixes being added continuously. Kernel developers are not committed to maintaining backward compatibility for internal interfaces. Thus, source code changes are frequent and inevitable, causing the absence or variation of a construct across kernel versions.

Kernel Configuration. Linux kernel configuration options allow users to customize various aspects of the kernel for different use cases. Architecture is one of the first and most important configuration options. Other options control various aspects of the kernel, including hardware support (e.g., NUMA), memory management (e.g., page size), and kernel features (e.g., cgroup). These options change the presence and definition of kernel constructs, thus affecting the dependency surfaces. The values of these options are determined by kernel developers and distribution maintainers; how the configuration affects the dependency surfaces could be blind spots for eBPF program developers and users.

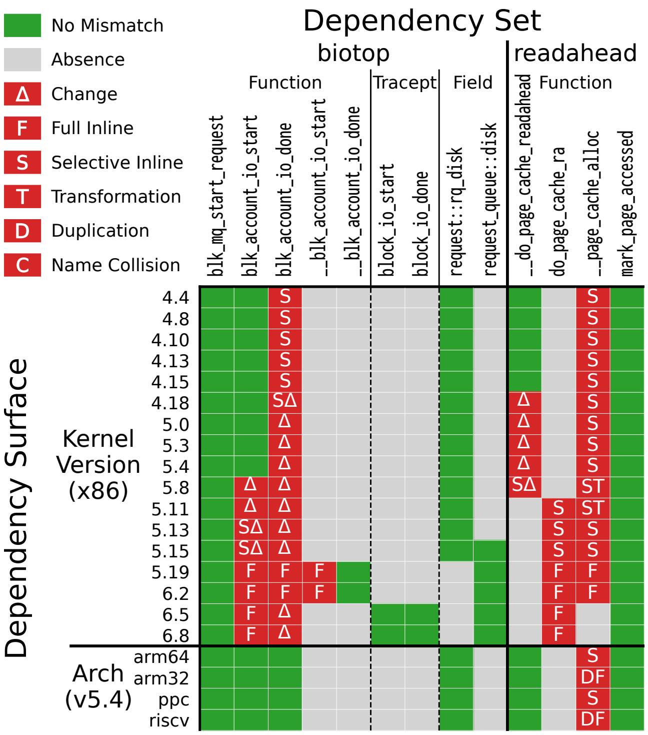
Kernel Compilation. After configuration, the compiler compiles the kernel image, applying various optimizations that can alter the presence and signature of functions. These optimizations, driven by complex compiler internals, are often opaque to users, making their effects difficult to predict and understand. Moreover, changes in the source code itself can lead to different optimization outcomes. This adds another layer of complexity, as seemingly minor changes can result in unexpected differences in the compiled image.

Selection of Kernel Images. We focus on kernels built by Ubuntu because of their widespread adoption and popularity [124]. This study includes 25 kernel images, covering 17 kernel versions over 8 years, 5 architectures (`x86`, `arm64`, `arm32`, `ppc`, and `riscv`), 5 configuration flavors (generic, low-latency, AWS, Azure, and GCP), and 14 compiler versions. Among the 17 kernel versions, 5 are released with long-term support (LTS)[2]: v4.4, v4.15, v5.4, v5.15, and v6.8 (for Ubuntu 16.04, 18.04, 20.04, 22.04, and 24.04). To ensure the accuracy of the data, we source kernel images and associated debug information from the `linux-image-*-dbgsym` packages built and maintained by Ubuntu [167].

3.3 Usage Scenarios

DepSurf offers valuable support for both development and maintenance of eBPF programs. When developing new eBPF programs or adding new dependencies to existing ones, DepSurf enables efficient compatibility checks across diverse kernel images. This allows developers to make informed decisions about which kernel constructs to rely upon. Furthermore, it reduces the risk of both explicit errors and silent failures that might otherwise go unnoticed. For maintenance,

[2] Ubuntu maintains its own schedule for LTS support, which differs from the schedule of the upstream Linux kernel team.



		biotop Function					biotop Tracept		biotop Field		readahead Function			
		blk_mq_start_request	blk_account_io_start	blk_account_io_done	__blk_account_io_start	__blk_account_io_done	block_io_start	block_io_done	request::rq_disk	request_queue::disk	__do_page_cache_readahead	do_page_cache_ra	__page_cache_alloc	mark_page_accessed
Kernel Version (x86)	4.4	🟩	🟩	S	⬜	⬜	⬜	⬜	🟩	⬜	🟩	⬜	S	🟩
	4.8	🟩	🟩	S	⬜	⬜	⬜	⬜	🟩	⬜	🟩	⬜	S	🟩
	4.10	🟩	🟩	S	⬜	⬜	⬜	⬜	🟩	⬜	🟩	⬜	S	🟩
	4.13	🟩	🟩	S	⬜	⬜	⬜	⬜	🟩	⬜	🟩	⬜	S	🟩
	4.15	🟩	🟩	S	⬜	⬜	⬜	⬜	🟩	⬜	🟩	⬜	S	🟩
	4.18	🟩	🟩	SΔ	⬜	⬜	⬜	⬜	🟩	⬜	Δ	⬜	S	🟩
	5.0	🟩	🟩	Δ	⬜	⬜	⬜	⬜	🟩	⬜	Δ	⬜	S	🟩
	5.3	🟩	🟩	Δ	⬜	⬜	⬜	⬜	🟩	⬜	Δ	⬜	S	🟩
	5.4	🟩	🟩	Δ	⬜	⬜	⬜	⬜	🟩	⬜	Δ	⬜	S	🟩
	5.8	🟩	Δ	Δ	⬜	⬜	⬜	⬜	🟩	⬜	SΔ	⬜	ST	🟩
	5.11	🟩	Δ	Δ	⬜	⬜	⬜	⬜	🟩	⬜	⬜	S	ST	🟩
	5.13	🟩	SΔ	Δ	⬜	⬜	⬜	⬜	🟩	⬜	⬜	S	S	🟩
	5.15	🟩	SΔ	Δ	⬜	⬜	⬜	⬜	🟩	🟩	⬜	S	S	🟩
	5.19	🟩	F	F	F	🟩	⬜	⬜	⬜	🟩	⬜	F	F	🟩
	6.2	🟩	F	F	F	🟩	⬜	⬜	⬜	🟩	⬜	F	F	🟩
	6.5	🟩	F	Δ	⬜	⬜	🟩	🟩	⬜	🟩	⬜	F	⬜	🟩
	6.8	🟩	F	Δ	⬜	⬜	🟩	🟩	⬜	🟩	⬜	F	⬜	🟩
Arch (v5.4)	arm64	🟩	🟩	🟩	⬜	⬜	⬜	⬜	🟩	⬜	🟩	⬜	S	🟩
	arm32	🟩	🟩	🟩	⬜	⬜	⬜	⬜	🟩	⬜	🟩	⬜	DF	🟩
	ppc	🟩	🟩	🟩	⬜	⬜	⬜	⬜	🟩	⬜	🟩	⬜	S	🟩
	riscv	🟩	🟩	🟩	⬜	⬜	⬜	⬜	🟩	⬜	🟩	⬜	DF	🟩

Figure 4. Dependency set analysis of biotop (left) and readahead (right). Columns represent the set of dependencies used by the program; structs are flattened to fields. Not all fields are shown due to space. Each row represents a dependency surface of a kernel image. The top shows x86 images across 17 versions, and the bottom shows images from 4 additional architectures. Each cell indicates if there is a dependency mismatch on the kernel image. Gray indicates mismatches due to absence, and red signals other types of mismatches, with letters labeled showing the causes. Name collision (C) does not occur in the examined programs and kernels.

developers can leverage DepSurf to proactively verify compatibility with the latest kernels. For issues identified during kernel release candidates, eBPF developers can provide feedback to kernel developers before stable releases (e.g., 9c2136b).

To demonstrate DepSurf's practical utility, we analyze two eBPF programs: biotop [76] and readahead [77]. Figure 4 presents their analysis reports, providing a comprehensive view of dependency mismatches across kernel images.

Diagnose biotop. We revisit the example of biotop from Section 2.5. As shown in Figure 4 (left), biotop depends on 5 functions, 2 tracepoints, and multiple structs.

DepSurf shows that the kernel function blk_mq_request (1st column) has no mismatches across all examined images. We next consider the evolution of two pairs of functions: (__)blk_account_io_{start, done}. For biotop to function correctly, it needs to monitor one of these pairs. The signature of kernel function blk_account_io_start (2nd column) has been changed (Δ) with a parameter removed since v5.8 (b5af37a). Since v5.19 (be6bfe3), the function is fully inlined (F), corresponding to the attachment error mentioned earlier. DepSurf shows the full inline of __blk_account_io_start (4th column), explaining why the first fix attempt failed. DepSurf reveals that while tracepoints block_io_{start, done} were added in v6.5 (5a80bd0), this addition does not address issues in earlier versions like v5.19 and v6.2. For fields, DepSurf shows that request::rq_disk is changed to request_queue::disk in v5.15, and both fields coexist in that version.

Fix readahead. We use readahead to show how DepSurf helps to diagnose and fix dependency mismatches. The program readahead (right) depends on 4 functions, with 3 of them having mismatches. Given the report, we pinpoint the kernel commit or configuration option that causes the mismatch, and fix the program by providing fallbacks [181].

We first trace the evolution of the first two functions. The return type of __do_page_cache_readahead is changed in v4.18 (c534aa3). A refactor in v5.8 (2c68423) leads to selective inline (S), where the function is only inlined at some call sites. The function is then renamed to do_page_cache_ra in v5.11 (8238287). Later, in v5.18 (56a4d67), the function is marked as static which results in full inline; meanwhile, another function page_cache_ra_order (not in the figure) is exposed instead. To address the issue after v5.18, we modify the program to first attempt attaching to page_cache_ra_order, falling back to other functions if it is not available.

For function __page_cache_alloc, in v5.16 (bb3c579), it becomes a simple wrapper for filemap_alloc_folio, resulting in full inline (F) reported by DepSurf. We fix the mismatch by trying to attach to filemap_alloc_folio first.

DepSurf also reports the function duplication (D) and full inline on arm32 and riscv. This occurs because both images disabled CONFIG_NUMA, which results in the function being defined as static inline in a header file, and duplicated in multiple C source files that include the header.

Summary. We demonstrate that DepSurf helps diagnose nuanced dependency mismatches arising from the complex interplay of source code changes, configuration-dependent behavior, and intricacies of the compilation process. While some mismatches can be fixed through careful programming, others require enhancements to the eBPF infrastructure. We explore these potential improvements in Section 6.

3.4 Implementation

DepSurf is implemented in 4.7k lines of Python. The main challenge of DepSurf is to identify and extract the relevant information from binary files. Our in-depth research on kernel internals, debug information, and CO-RE has enabled us to develop a robust solution.

Dependency Surface Analysis. From a kernel image vmlinux in ELF format [3], DepSurf combines data from the DWARF debug sections [31, 32, 59], symbol table, and data sections [2] to extract the following information:

Functions. DepSurf parses the debug information to obtain function declarations, including the function name, parameter names, parameter types, and return type. To determine the inline status of a function, DepSurf examines the inline attribute in all instances of the function and checks its presence in the symbol table. DepSurf identifies transformed functions by detecting specific suffixes in their symbol names. To handle functions with the same names, DepSurf records the file path and line number, allowing it to distinguish between function duplications and name collisions.

Structs. DepSurf extracts struct definitions from the debug information. Named nested structs are treated as separate structs. Fields in anonymous nested structs are flattened into the parent struct. Array lengths are recorded and type quantifiers (e.g., `const`) are preserved.

Tracepoints. DepSurf extracts tracepoints from the array bounded by `__{start,stop}_ftrace_events`, following the same logic as the kernel. To statically extract this data without booting the kernel, we implemented a generic parser that interprets and dereferences contents in the data sections. The parser also handles architecture-specific details such as pointer size, endianness, and relocation.

System Calls. DepSurf reads the `sys_call_table` array for the addresses of system calls, and then reverse-looks up their names from the symbol table.

Dependency Set Analysis. From an eBPF object file, DepSurf parses section names to identify the hooks used by the program, such as kernel functions, tracepoints, and system calls [85]. Dependencies on structs and fields are extracted from the `.BTF.ext` section used for CO-RE relocation [3, 56, 85, 120]. For chained member access expressions (e.g., `a[1].b->c`), DepSurf records all intermediate structs and fields in the chain. Fields not directly accessed by the eBPF program are not recorded.

Usability and Performance. Users can easily incorporate new kernel images and eBPF programs by providing their paths. Extracting the dependency surface of a kernel image takes on average 104 seconds, with processing times ranging from 58 to 132 seconds[3]. Kernel images are processed in parallel. Computing the declaration differences across 17 kernel images completes in 3 seconds. Dependency set analysis of an eBPF program finishes within a fraction of a second.

4 Dependency Surface Analysis

This section presents our comprehensive analysis of dependency surfaces, leveraging the dataset generated by DepSurf from a diverse collection of kernel images. We begin with an overview of the dependency mismatches, consequences, and implications, followed by an in-depth analysis.

Dependency Mismatches. Table 1 summarizes our key findings on mismatches that can occur when eBPF programs use the kernel constructs. For each type of construct in the dependency surface, we report the causes of mismatches, their frequencies, and the consequences. We identify three contributors that shape the dependency surface:

[3] Measured on an M1 Macbook Air running a Ubuntu 24.04 virtual machine.

	Type	Cause	Freq	Consequence
Source Code	Function	Addition/Removal	24%/10%	Attachment Error
		Change	6%	Stray Read
	Struct	Addition/Removal	24%/4%	Compilation Error
		Change	18%	Stray Read or CE
	Tracept	Addition/Removal	39%/5%	Attachment Error
		Change	16%	Stray Read or CE
Configuration	Function	Addition/Removal	26%/25%	Attachment Error
		Change	0.3%	Stray Read
	Struct	Addition/Removal	24%/22%	Compilation Error
		Change	1.8%	Stray Read or CE
	Tracept	Addition/Removal	8%/34%	Attachment Error
	Syscall	Availability	by arch	Attachment Error
		Traceability	by arch	Missing Invocation
	Register	Difference	by arch	Relocation Error
Compilation	Function	Full Inline	36%	Attachment Error
		Selective Inline	11%	Missing Invocation
		Transformation	16%	Attachment Error
		Duplication	12%	Missing Invocation
		Name Collision	0.6%	Stray Read

Table 1. Summary of dependency mismatches. The percentages in the "Freq" column represent (1) for source code: the maximum differences between two consecutive LTS versions; (2) for configuration: the maximum differences compared to the generic `x86` kernel for v5.4; (3) for compilation: the fractions of affected functions. All compilation errors (CE) also imply relocation errors.

Consequence	Implication
Compilation Error (CE) ⇒ Relocation Error	Explicit Error (before execution)
Attachment Error	
Stray Read	Incorrect Result (might be detectable)
Missing Invocation	Incomplete Result (difficult to detect)

Table 2. Implications for the consequences in Table 1 (last column).

Kernel source code (§4.1). DepSurf's analysis across kernel versions reveals substantial changes in the presence and definition of constructs. Between two LTS releases, the addition of new constructs is significant: up to 24% for functions and structs, and 39% for tracepoints. Conversely, removals occur at rates of 10%, 4%, and 5% respectively. Both additions and removals lead to the absence of constructs between kernel versions. Moreover, existing constructs often change between versions. Our analysis shows that 6% of functions, 18% of structs, and 16% of tracepoints experience changes in their definitions between LTS releases.

Kernel configuration (§4.2). Across kernel configurations, DepSurf finds that the presence of many constructs is affected: up to 26% of functions, 24% of structs, and 8% of tracepoints are added, and 25%, 22%, and 34% are removed respectively. In contrast, changes to existing constructs are minimal: only 0.3% of functions and 1.8% of structs are modified, with no changes observed in tracepoints. Furthermore, system call availability and traceability vary depending on the architecture. Architecture-specific register layouts impact eBPF programs that access function arguments.

Kernel compilation (§4.3). DepSurf's analysis of the compilation process reveals several mismatches related to functions: 36% of them are fully inlined, whereas 11% are inlined only on some call sites. 16% of the functions are transformed by the compiler. Function duplication and name collisions affect 12% and 0.6% of the functions respectively.

Takeaway 1: Dependency surfaces are inherently unstable, driven by the evolving source code, diverse configuration options, and intricacies in the compilation process.

Consequences and Implications. Table 2 presents the consequences and implications of the dependency mismatches.

Explicit error. Compilation errors, relocation errors, and attachment errors are explicit errors reported before an eBPF program's execution. These mismatches primarily stem from the absence of necessary constructs in the kernel image. For example, a missing field triggers a compilation error, or causes a relocation error when loading a once-compiled program on a different kernel. If the expected hook (e.g., function, tracepoint, or system call) is absent, an attachment error is raised. A construct may be absent for several reasons. For example, the kernel image predates the version where the construct was added, or postdates the version where the construct was removed. For functions specifically, absence can also result from compiler optimizations such as full inline and transformation. These explicit errors serve as clear indicators of mismatches between the eBPF program and the target kernel version.

Incorrect result. Changes to function signatures or struct field types can lead to stray reads. Since eBPF programs access function arguments through untyped registers, changes in a function's signature lead to the program reading incorrect data. Similarly, when accessing struct fields, if a field's type changes to one compatible with the old type (e.g., `char` to `int`), the program can still compile and run but misinterpret the data. Although some runtime checks may detect some data validity issues, many errors still propagate undetected. For example, passing invalid addresses to `bpf_probe_read` returns an error code, which is often unchecked by developers.

Incomplete result. Missing invocations can lead to seemingly reasonable but incomplete results. This can arise from several causes: 1) Selective inline, where a function is inlined at some call sites but not others, resulting in missed invocations at the inlined locations. 2) Function duplication, where only the first instance of a duplicated function is attached, missing subsequent ones. 3) Incomplete tracing support for certain system calls. These missing invocations pose a significant challenge in terms of detection and diagnosis, as the partial results may appear valid despite being incomplete.

Takeaway 2: Undetected dependency mismatches can lead to silent incorrect or incomplete results during execution.

		Function				Struct				Tracepoint			
		#	+%	−%	Δ%	#	+%	−%	Δ%	#	+%	−%	Δ%
LTS (2 yr)	4.4	36k				6.2k				502			
			24	7	5		24	4	18		39	5	8
	4.15	42k				7.3k				675			
			21	8	4		19	4	15		15	3	8
	5.4	48k				8.4k				752			
			22	10	5		17	6	16		14	5	16
	5.15	54k				9.3k				818			
			23	8	6		16	4	16		18	4	14
	6.8	62k				10.5k				932			
Regular Releases (6 mo)	4.4	36k				6.2k				502			
			8	3	2		9	2	9		7	-	5
	4.8	38k				6.6k				539			
			4	2	1		3	1	6		5	1	3
	4.10	39k				6.8k				559			
			7	3	2		5	1	9		15	1	2
	4.13	41k				7.1k				635			
			5	1	1		5	1	5		9	2	3
	4.15	42k				7.3k				675			
			9	3	2		5	2	7		4	3	1
	4.18	44k				7.6k				683			
			4	3	2		4	2	7		4	0.4	3
	5.0	45k				7.8k				704			
			6	2	1		6	1	7		5	0.1	3
	5.3	47k				8.2k				737			
			3	1	1		3	1	3		2	-	0.3
	5.4	48k				8.4k				752			
			14	6	2		6	3	8		6	2	7
	5.8	52k				8.6k				785			
			7	2	2		6	2	7		4	1	3
	5.11	54k				9.0k				813			
			3	5	1		4	1	4		1	2	2
	5.13	53k				9.2k				805			
			3	2	1		3	1	5		2	0.5	6
	5.15	54k				9.3k				818			
			7	3	2		4	2	7		6	3	6
	5.19	56k				9.6k				843			
			5	3	2		4	1	6		4	0.4	4
	6.2	57k				9.8k				871			
			6	1	2		3	1	6		6	1	5
	6.5	59k				10.0k				917			
			6	2	1		5	0.5	6		2	0.1	2
	6.8	62k				10.5k				932			

Table 3. Kernel source code difference reported by DepSurf. We show the number (#) in each version, and the percentage added (+), removed (−) and changed (Δ) between versions. Bar length is proportional to the percentage, normalized to the column's maximum.

4.1 Kernel Source Code

To analyze the effect of kernel source code changes on the dependency surface, we examine 17 kernel versions spanning 8 years, from v4.4 to v6.8 (Ubuntu 16.04 to 24.04). We focus on three types of constructs: functions, structs, and tracepoints.

Kernel Functions. eBPF programs can attach to kernel functions via kprobes, reading their arguments and return value. We analyze the addition, removal, and changes of kernel functions, as shown in the first column group in Table 3.

Additions and removals. The Linux kernel has over 35k functions, with the number increasing by roughly 1k to 4k every six months, as shown by the first "#" column in Table 3. The "+%" and "−%" columns show the percentage of functions removed or added from one version to the next (e.g., 7% of existing functions are removed when updating to v4.15, and 24% more functions are added). One example of function addition/removal is the introduction of the `folio` concept (`49f8275` in v5.16) to efficiently represent contiguous pages, renaming hundreds of functions [172]. Absence of a function results in an attachment error.

Changes. Function signature changes are slightly lower than function additions and removals, ranging from 1% to 2% between each regular release, and 5% on average across LTS versions. Since eBPF programs attached to kprobes read function arguments from registers, and manually cast them to the expected types (e.g., line 4 in Listing 1), function signature changes do not result in an explicit type error. Instead, they cause silent stray reads, leading to incorrect results. Listing 3

		Linux Kernel Version			
		4.4 - 4.15	4.15 - 5.4	5.4 - 5.15	5.15 - 6.8
Function	No. changed	2.0k	1.8k	2.2k	3.0k
	- Param added	57%	53%	60%	51%
	- Param removed	41%	36%	44%	48%
	- Param reordered	19%	19%	25%	19%
	- Param type changed	26%	23%	26%	25%
	- Return type changed	15%	21%	13%	17%
Struct	No. changed	1.1k	1.1k	1.3k	1.5k
	- Field added	72%	74%	75%	74%
	- Field removed	41%	40%	40%	42%
	- Field type changed	37%	34%	32%	34%
Tracept	No. changed	39	54	119	115
	- Event changed	95%	81%	86%	92%
	- Func changed	46%	54%	32%	51%

Table 4. Breakdown of kernel source code changes.

```
1 int f(int a, double b);         // original
2 int f(int a, double b, int c); // parameter added
3 int f(int a);                   // parameter removed
4 int f(double b, int a);         // parameter reordered
5 int f(int a, int b);            // parameter type changed
6 double f(int a, double b);      // return type changed
```

Listing 3. Examples of function signature changes.

shows examples of function signature changes, with their frequencies shown in Table 4.

First, parameter addition and removal (lines 2, 3) are the most common, occurring for about 50% and 40% of all function changes across LTS versions. As an example, vfs_rename was changed from taking 6 arguments to a single renamedata struct in 9fe6145: an eBPF program written assuming the first signature will, for example, fail to read the inode number from the first argument, and instead read random data.

Second, changes to parameter type or return type (lines 5, 6) are less common, accounting for about 25% and 15% of all function changes, respectively. Such type changes usually imply a semantic change. For example, in 18b43a9, account_idle_time was changed from taking cputime_t (representing time in jiffies) to u64 in nanoseconds.

Third, reordering of parameters (line 4) is less common, accounting for about 20% of all changes; in this case, only the indexes of an argument are changed. For example, in the case where an argument is inserted at the beginning of vfs_create in 6521f89, the original four arguments are considered as reordered.

Takeaway 3: Function signature changes are common, causing eBPF programs to silently read wrong data.

Special Kernel Functions. In addition to regular functions, we discuss two special kinds of kernel functions that enable eBPF programs to change kernel behavior.

LSM Hooks. Linux Security Module (LSM) hooks serve as security checkpoints in the kernel, invoked to determine if specific operations are permitted [61]. eBPF programs attached to LSM hooks can control kernel behavior by their return values. This powerful feature can be used to implement security policies or patch security vulnerabilities [29, 94, 105, 106]. Despite their significance, LSM hooks are not stable [146, 151], a fact even overlooked by some kernel developers [108, 142].

The Linux kernel contains over 150 LSM hooks; on average, 9% of LSM hooks are added and 2% removed across each LTS version. Usually, a hook is removed because it is no longer needed or has been replaced by another hook (e.g., 3cf2993, da2441f). Changes to an LSM hook's signature have serious security implications. For example, stray reads can cause a program to mistakenly allow denied operations.

Kfuncs. Similar to eBPF helper functions, kfuncs are kernel functions callable from eBPF programs, but they lack a stable interface [55]. There are around 100 kfuncs in v6.8. We did not observe any kfuncs changing signature in the kernels we studied, but some kfuncs have been removed (e.g., f85671c, 6499fe6) or renamed (e.g., d2dcc67). Kfuncs are registered to the verifier, so programs misusing them will be rejected.

Kernel Structures. eBPF programs invoke helper functions provided by the kernel, such as bpf_probe_read, to read kernel structures (e.g., line 5 in Listing 1). We show the numbers of addition, removal, and changes of kernel structures in Table 3, and breakdown the changes in Table 4.

Additions and removals. The Linux kernel contains a large number of defined structures: between 6k and 10k per kernel version. As expected, the number of structures in the kernel increases over time. Table 3 shows that while the absolute number of structures is significantly less than the number of functions in the kernel, the percentages that are added or deleted over time are relatively similar to one another: about 20% structs are added and 5% removed across each LTS version. For example, the renaming from compat_time to old_time32 in 9afc5ee adds and removes 3 structs. If a struct is used by an eBPF program, but absent in the kernel, the eBPF program will fail to compile, or a once-compiled program will fail to load with a relocation error.

Changes. Struct fields can be added or removed, accounting for around 60% and 35% of the field changes, respectively. As an example, task_struct's long state field was changed to unsigned int __state in 2f064a5 with a later commit (5616e89) fixing an in-tree eBPF program. The absence of fields leads to the same compilation errors or relocation errors. The type of a field in a struct can also be changed, accounting for about 30% of all struct changes. For example, the change from cputime_t to u64 affects many structs in the kernel, including the utime field in task_struct in 5613fda. If the changed type is not compatible with the original type, a compilation error or relocation error will be reported. Otherwise, the old type will be used, risking reading wrong data.

Takeaway 4: Most struct changes are detectable by compilers, but compatible type changes can cause silent errors.

Tracepoints. A tracepoint is a static instrumentation point in the Linux kernel that allows the kernel to record events. Tracepoints are desired (and sometimes assumed) to be stable by users, but as we will show, addition, removal, and changes are still surprisingly common.

Additions and removals. Table 3 shows that there are over 500 tracepoints, orders of magnitude fewer than the number of functions or structures. As the kernel evolves, tracepoints need to evolve as well. Across each LTS version, approximately 25% new tracepoints are added while 4% are removed. However, tracepoints are sometimes added or removed unnecessarily. For example, 11e9734 removed the original tracepoint `kmem_alloc` and renamed `kmem_alloc_node` to take its place because “maintaining both does not make sense” even though the change “will break some tools” [84]. This decision contradicts the longstanding principle of not breaking userspace, highlighting the tension between kernel evolution and maintaining backward compatibility.

Changes. Table 3 shows that about 8% to 16% of tracepoints are changed across each LTS version. To better understand these changes, we need to delve into the two key components of a tracepoint: the event struct and the tracing function. When the tracing function is called in the kernel, it generates an event struct, which is then passed to eBPF programs attached as an argument. Alternatively, eBPF programs can directly attach to the tracing function itself, a method known as raw tracepoint, to bypass the event struct generation. It has been made clear in the mailing list that the tracing functions are not stable [139].

Table 4 breaks down the tracepoint changes. Most tracepoint changes, ranging from 81% to 95%, include event struct changes. For example, the trace event for `itimer_state` used to have one field, `value_usec`, which was changed to `value_nsec` in bd40a17. About 45% of tracepoint changes include changes in the tracing function, significantly lower than the trace event changes, despite not being promised to be stable. For example, a54895f removed the first argument of `block_rq_issue`, requiring multiple tools to be updated to work with the new kernel [133, 134].

Takeaway 5: Tracepoints are not as stable as presumed.

4.2 Kernel Configuration

Before the source code is compiled into a binary, the kernel is configured with a set of options. To understand the impact of the configuration options on the dependency surfaces, we compare Linux v5.4 with `x86` architecture and generic flavor to 4 other architectures (`arm64`, `arm32`, `ppc`, `riscv`) and 4 flavors (low-latency, AWS, Azure, GCP).

Table 5 shows the number of configuration options in each (8.8k options in `x86`) and the impact on different kernel constructs. The configuration options differ significantly across architectures, whereas the low-latency flavor is almost identical to the generic flavor (except for minor tweaks for the scheduler and timer); the AWS and Azure flavors aggressively remove device drivers not needed in the cloud.

		def-		Architecture				Flavor		
		ault	`arm64`	`arm32`	`ppc`	`riscv`	AWS	Azure	GCP	low-lat
Config	#	8.8k	9.6k	9.6k	8.1k	7.6k	6.4k	5.3k	8.6k	8.8k
Func	#	48.0k	49.4k	48.8k	42.8k	36.6k	46.5k	45.5k	48.1k	48.0k
	+	-	9.2k	12.6k	5.4k	2.1k	328	992	450	57
	–	-	7.9k	11.8k	10.6k	13.5k	1.8k	3.5k	319	41
	Δ	-	120	106	137	101	2	10	1	-
Struct	#	8.4k	9.1k	8.6k	7.4k	6.6k	8.0k	7.8k	8.4k	8.4k
	+	-	1.7k	2.0k	570	157	83	257	68	4
	–	-	1.0k	1.9k	1.6k	2.0k	483	833	123	1
	Δ	-	81	154	116	98	19	28	14	5
Tracept	#	752	685	690	648	625	747	739	752	752
	+	-	45	70	25	-	4	26	-	-
	–	-	112	132	129	127	9	39	-	-
	Δ	-	-	-	-	-	-	-	-	-
Native Syscall	#	333	291	378	347	280	333	333	333	333
	+	-	2	74	23	2	-	-	-	-
	–	-	44	29	9	55	-	-	-	-
Register	Δ	-	Yes	Yes	Yes	Yes	-	-	-	-

Table 5. Configuration differences compared to generic `x86` kernel for v5.4. Bar length normalized to the maximum for each row group.

The configuration options change the dependency surfaces of the kernel by conditionally compiling different parts of the kernel. The presence of a function depends heavily on the configuration options. However, function signatures are rarely changed since different configurations provide different implementations for the same function definition. A tiny fraction of structs are changed between architectures and flavors. For example, `task_struct`, having more than 80 `#ifdefs`, is often changed. Configuration also affects the presence of tracepoints, but no tracepoints changes are observed.

Takeaway 6: Configuration mainly influences the presence of kernel constructs, but rarely changes the definitions.

System Call. The support for system calls varies across architectures. Our analysis focuses on two critical aspects of system calls: the availability of architecture-native system calls and the traceability of 32-bit compatible system calls.

Availability. Table 5 shows that the availability of native system calls varies across architectures. Notably, the `arm64` architecture lacks 44 system calls that are present in `x86`. Newer architectures are designed to exclude redundant system calls when better alternatives are available. For example, `open` and `chmod` are replaced by their `*at` counterparts, and `[v]fork` by `clone`. Blindly attaching to these system calls will cause attachment error.

Traceability. While most 64-bit architectures support 32-bit system calls for backward compatibility, tracing these calls presents a significant challenge. Many architectures, including `x86`, `arm64`, and `riscv`, lack native tracing support for these 32-bit system calls. This creates a critical blind spot in

```
1 // File: fs/sync.c
2 int vfs_fsync() { /* logic */ }         // func definition
3 long sys_fsync() { vfs_fsync(); }       // inlined call site
4 // File: fs/aio.c
5 int vfs_fsync();                        // func declaration
6 void aio_fsync_work() { vfs_fsync(); } // regular call site
```

Listing 4. Example of selective inline in Linux kernel.

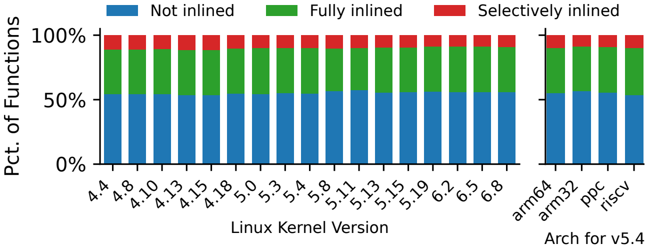


Figure 5. Percentages of functions fully and selectively inlined.

system monitoring and security enforcement. Consequently, a malicious program can deliberately use 32-bit system calls to evade detection.

Register Layout. eBPF programs using kprobes access function arguments through the architecture-specific struct pt_regs (e.g., line 4 in Listing 1). For example, the first argument is accessed with ctx->di on x86, but ctx->regs[0] on arm64. Macros like PT_REGS_* were introduced to abstract the register access, improving portability. Unfortunately, CO-RE is not compatible with macros, causing a relocation error if the eBPF program runs on an architecture other than the one it was compiled for.

4.3 Kernel Compilation

The compilation process adds another layer of complexity to the functions in the dependency surfaces. We analyze the impact of function inline, transformation, duplication, and name collisions on kernel images across 14 GCC versions, 17 kernel versions, and 5 architectures.

Function Inline. Compiler optimization performs inline by copying the function body to the call site. It causes two issues for eBPF programs: if the function is *fully inlined*, it cannot be attached by eBPF programs as the function is absent in the symbol table; if the function is *selectively inlined* at some but not all call sites, eBPF programs attached to the function will only be invoked at the non-inlined call sites.

Full inline. The compiler can fully inline a function when all these conditions are met: (1) the function is marked as static, (2) the address of the function is not taken, (3) the function definition is visible to the compiler, and (4) the function size is within the inline threshold. The compiler has discretion over inline decisions, regardless of the inline keyword.

Selective inline. Over 3k kernel functions are selectively inlined. Listing 4 shows an example of selective inline in the Linux kernel. Lines 1 to 3 show that vfs_fsync is called by sys_fsync within the same file, and is inlined by the compiler. However, for other call sites such as aio_fsync_work, the compiler only sees the function declaration of vfs_fsync, and cannot inline the function (lines 4-6). If an eBPF program attaches to vfs_fsync, the attachment will succeed, but miss inlined call sites issued by the system calls, leading to incorrect behavior. Determining if a function is selectively inlined is difficult, requiring examination of the assembly code at each call site.

```
1 int f              (int a, int* b) { return a + *b; }
2 int f.constprop.0        (int* b) { return 1 + *b; }
3 int f.constprop.0.isra.0(int  b) { return 1 +  b; }
```

Listing 5. Example of function f transformed by compiler.

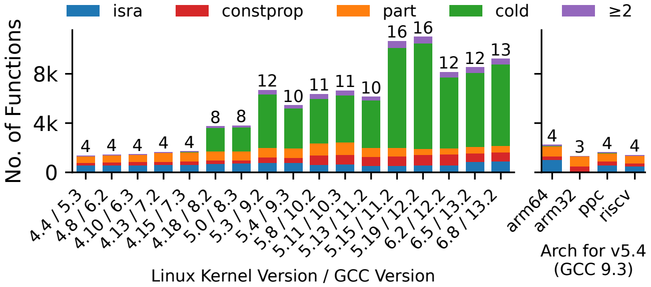


Figure 6. Function transformation. The percentages are labeled on the top of the bars. ≥2 denotes functions transformed by 2 or more optimizations.

Figure 5 shows that, across the kernel versions and architectures, 32% to 36% of functions are fully inlined, 9% to 11% of functions are selectively inlined; the variation is permanent, but not in a high percentage (4% and 2%). There is no guarantee that the same inline decision will be made across different compiler versions, optimization flags, kernel versions, or configurations.

Takeaway 7: Selective inline is surprisingly common, causing silent error with deceptive data.

Function Transformation. Compiler optimizations can also transform function signatures and names, causing eBPF programs to fail to attach and read the correct arguments. For example, in line 2 of Listing 5, interprocedural constant propagation [19, 67, 170] replaces argument a with a constant and adds suffix constprop to the function name. In line 3, interprocedural scalar replacement of aggregates (ISRA) [66, 170] changes argument type from int* to int and adds suffix isra. Other transformations include function splitting (with suffix part), and hot/cold function partitioning (with cold).

Figure 6 shows that up to 16% of functions in the symbol table are transformed by the compiler, depending on the compiler version and optimization flags. For example, due to default optimizations in GCC 8 (and thus in Linux 4.18 with GCC 8.2), functions with cold appear; arm32 does not have any function with isra since it is disabled in a077224.

Function transformation can be detected when an attachment fails, and a function with a suffix is in the symbol table. However, the compiler does not expose how the function arguments are changed, making it difficult to read the correct arguments in the eBPF programs.

	Linux Kernel Version				
	4.4	4.15	5.4	5.15	6.8
Unique Global	17.2k	20.1k	22.7k	26.6k	31.5k
Unique Static	35.7k	41.7k	48.2k	53.3k	60.2k
Static Duplication	4.0k	4.8k	5.5k	6.2k	7.4k
Static-Static Collision	404	398	411	444	498
Static-Global Collision	10	26	27	26	29

Table 6. Function duplication and name collision.

Functions with the Same Name. During the compilation process, the linker enforces unique names for global functions, but not for static ones. Meanwhile, static functions also end up in the symbol table, leading to ambiguity in identifying the correct function for eBPF programs to attach to. We categorize the issue into two cases:

Function duplication. When a static function is defined in a header file, the same function is duplicated in every file that includes that header. Table 6 shows there are up to 7k of these cases. For example, `get_order` defined in `include/asm-generic/getorder.h` is included in 1125 files in v5.4. Attaching to only the first function causes some invocations to be missed by eBPF programs, but naively attaching to all also causes problems, as described next.

Name collision. There are about 400 cases where a static function shares the same name as another static function and 20 cases with a global one. In some cases, the functions serve similar purposes; for example, `destroy_inodecache` is defined by 16 different filesystems. In other cases, two unrelated functions happen to share the same name; for example, `do_readahead` is defined with different signatures in `fs/jbd2/recovery.c` and `mm/readahead.c`. Attaching to all these unrelated functions is unlikely to be the correct behavior.

5 Dependency Set Analysis

We use DepSurf to perform dependency set analysis on 53 eBPF programs from BCC [75] and Tracee [143]. BCC [75] is a well-known project for Linux kernel observability and is used in production by major companies [74]. These programs are specialized to monitor a specific kernel subsystem, including CPU and processes (15), memory (5), storage (18), and network (11). Tracee [143] is a more complex program that uses eBPF for runtime security. It monitors various kernel subsystems, including storage and network, to detect security threats.

Dependency Set. The Σ columns in Table 7 show the total number of dependencies for each program. For example, Tracee depends on 67 functions, 99 structs, 254 fields, 13 tracepoints, and 446 system calls. Table 8 shows a summary for all 53 analyzed programs; the Σ rows reveal diverse dependency types across the programs: 25 programs (47%) depend on functions; 42 programs (79%) rely on structs and fields; 25 (47%) are attached to tracepoints; and 8 of them (15%) interact with system calls. Only 11 programs do not depend on any structs or fields (e.g., `vfsstat`), simply monitoring the

Program	Function							Struct		Field			Tracept			Syscall	
	Σ	∅	Δ	F	S	T	D	Σ	∅	Σ	∅	Δ	Σ	∅	Δ	Σ	∅
Tracee	67	14	16	5	14	14	2	98	14	250	53	9	13	3	4	446	202
klockstat	14	3	-	-	4	-	-	-	-	-	-	-	-	-	-	-	-
vfsstat	8	-	5	-	6	1	-	-	-	-	-	-	-	-	-	-	-
biotop	5	2	2	3	2	-	-	3	-	7	2	1	2	2	-	-	-
cachestat	5	2	2	-	1	-	-	-	-	-	-	-	2	2	1	-	-
fsdist	5	2	1	-	2	2	-	-	-	-	-	-	-	-	-	-	-
tcptracer	5	-	1	-	-	3	-	6	-	14	-	-	-	-	-	-	-
readahead	4	3	1	2	3	1	1	2	1	1	1	-	-	-	-	-	-
fsslower	4	1	-	-	2	1	-	5	-	6	-	-	-	-	-	-	-
filelife	4	-	3	-	2	-	-	5	1	6	2	-	-	-	-	-	-
biostacks	3	1	2	2	3	-	-	3	-	5	2	-	2	2	-	-	-
tcpconnlat	3	-	-	-	-	2	-	4	1	11	1	-	1	1	1	-	-
numamove	2	2	-	1	-	-	-	-	-	-	-	-	-	-	-	-	-
biosnoop	2	1	1	1	2	-	-	3	-	9	2	1	4	1	3	-	-
filetop	2	-	-	-	2	-	-	6	-	10	-	-	-	-	-	-	-
tcpsynbl	2	-	-	-	-	2	-	1	-	2	-	-	-	-	-	-	-
tcpconnect	2	-	-	-	-	1	-	3	-	8	-	-	-	-	-	-	-
bindsnoop	2	-	-	-	-	-	-	5	-	14	4	1	-	-	-	-	-
tcptop	2	-	-	-	-	-	-	3	-	9	-	-	-	-	-	-	-
oomkill	1	-	1	-	1	1	-	3	1	4	2	-	-	-	-	-	-
capable	1	-	1	-	1	1	-	-	-	-	-	-	-	-	-	-	-
tcprtt	1	-	1	-	-	1	-	6	-	12	-	-	-	-	-	-	-
mdflush	1	-	1	-	-	1	-	3	-	4	2	-	-	-	-	-	-
solisten	1	-	-	-	-	1	-	5	-	8	-	-	-	-	-	-	-
slabratetop	1	-	-	-	-	-	-	1	-	2	-	1	-	-	-	-	-
memleak	-	-	-	-	-	-	-	11	9	17	14	-	10	4	7	-	-
tcppktlat	-	-	-	-	-	-	-	7	1	12	-	-	3	3	3	-	-
mountsnoop	-	-	-	-	-	-	-	7	1	6	-	-	-	-	-	2	-
runqlat	-	-	-	-	-	-	-	5	-	11	3	1	3	-	3	-	-
tcpstates	-	-	-	-	-	-	-	4	1	13	7	1	1	1	1	-	-
runqlen	-	-	-	-	-	-	-	4	-	5	-	-	-	-	-	-	-
biolatency	-	-	-	-	-	-	-	3	-	7	2	1	3	-	3	-	-
bitesize	-	-	-	-	-	-	-	3	-	6	2	-	1	-	1	-	-
sigsnoop	-	-	-	-	-	-	-	3	-	5	-	-	1	-	1	3	-
execsnoop	-	-	-	-	-	-	-	3	-	4	-	-	-	-	-	1	-
biopattern	-	-	-	-	-	-	-	2	2	6	6	-	1	-	1	-	-
tcplife	-	-	-	-	-	-	-	2	1	12	10	1	1	1	1	-	-
syscount	-	-	-	-	-	-	-	2	-	4	-	-	2	-	-	-	-
statsnoop	-	-	-	-	-	-	-	2	-	2	-	-	-	-	-	5	4
opensnoop	-	-	-	-	-	-	-	2	-	2	-	-	-	-	-	2	1
futexctn	-	-	-	-	-	-	-	2	-	2	-	-	-	-	-	1	-
profile	-	-	-	-	-	-	-	1	1	1	1	1	-	-	-	-	-
llcstat	-	-	-	-	-	-	-	1	1	1	1	-	-	-	-	-	-
offcputime	-	-	-	-	-	-	-	1	-	6	2	-	1	-	1	-	-
runqslower	-	-	-	-	-	-	-	1	-	5	2	-	3	-	3	-	-
cpudist	-	-	-	-	-	-	-	1	-	5	2	-	1	-	1	-	-
wakeuptime	-	-	-	-	-	-	-	1	-	4	-	-	2	-	2	-	-
exitsnoop	-	-	-	-	-	-	-	1	-	4	-	-	1	-	-	-	-
hardirqs	-	-	-	-	-	-	-	1	-	1	-	-	2	-	-	-	-
drsnoop	-	-	-	-	-	-	-	-	-	-	-	-	2	-	1	-	-
softirqs	-	-	-	-	-	-	-	-	-	-	-	-	2	-	-	-	-
cpufreq	-	-	-	-	-	-	-	-	-	-	-	-	1	-	-	-	-
syncsnoop	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	6	1

Table 7. Dependency set analysis of 53 eBPF programs. For each program, we show the total number of dependencies (Σ) and the number of dependencies with mismatches, including absence (∅), change (Δ), full inline (F), selective inline (S), transformation (T), and duplication (D). Tools free of mismatches are highlighted in blue.

		# Programs	# Uniq Deps	Reported Issue No.
Func	Σ	25	126	–
	∅	10	29	3687, 3695, 3692, 3747, 4337, 4885
	Δ	14	31	1911, 3360, 4339, 4340, 4346
	F	6	11	4261, 4478, 4638, 4700, 5115
	S	14	32	703, 1667, 2252, 2373, 3913
	T	14	28	1754, 3293, 3339, 3315, 4937
	D	2	3	–
Struct	Σ	43	135	–
	∅	13	31	4340, 4339
Field	Σ	43	342	–
	∅	22	102	1384, 3612, 3647, 3650, 3658, 3672 3680, 3859, 3903, 3926, 3936
	Δ	10	13	3845, 3865
Tracept	Σ	25	44	–
	∅	10	15	1636, 2816, 4384
	Δ	18	23	3317, 3338, 4076, 4476
Syscall	Σ	8	448	–
	∅	4	204	3012, 3668, 3843, 4287

Table 8. Summary of Table 7. For each type of construct, we show the numbers of programs and unique dependencies for the construct (Σ), and those affected by absence (∅), change (Δ), full inline (F), selective inline (S), transformation (T), and duplication (D). We also list the GitHub issue numbers that manifest the mismatches.

number of invocations. In total, they depend on 126 unique functions, 134 structs, 345 fields, 44 tracepoints, and 448 system calls, with most of the system calls used by Tracee. We observe that although an equal number of programs attach to tracepoints and functions, the number of unique tracepoints used (44) is much less than that of functions (126).

Takeaway 8: eBPF programs exhibit diverse dependency sets.

Dependency Mismatches. For each dependency used by the program, we examine dependency mismatches across 21 kernel images as used in Figure 4. Our analysis reveals that only 9 of the programs (highlighted in blue in Table 7) are free from mismatches across all kernel images; notably, these tools generally have simpler functionality. For the mismatches found by DepSurf, we further cross-referenced the GitHub issues reported to the BCC project (shown in Table 8), providing concrete examples of real-world dependency mismatches encountered by developers.

Table 8 shows the summary of dependency mismatches with the 53 eBPF programs we analyzed. Of the 25 programs that depend on functions, 3 main types of mismatches affect 14 programs each: seletive inline, leading to incomplete results; signature changes, causing incorrect results; and compiler transformation, resulting in attachment errors. Among the 42 programs that depend on structs or fields, the most common mismatches are their absence, which causes compilation errors. For the 25 programs that depend on tracepoints, changes to tracepoints affect 18 programs. 8 programs depend on 448 unique system calls with about half of them unavailable on some kernel images, affecting 4 programs.

Takeaway 9: Dependency mismatches are widespread in eBPF programs, requiring continuous maintenance efforts.

6 Discussion

We now examine the challenges and potential solutions for dependency mismatches, followed by our limitations.

Inherent Challenges. The fundamental challenges of dependency mismatches stem from the unstable nature of Linux kernel internals. As evolution of the kernel internals is inevitable, eBPF programs that depend on them should evolve as well. It is extremely challenging for any tool to automatically adapt to arbitrary changes to the kernel internals – some manual effort is always necessary.

DepSurf helps reduce this maintenance burden by providing a systematic approach to detect and analyze dependency mismatches. We believe that our work represents an important first step towards a more stable eBPF ecosystem by raising awareness and demonstrating the importance of dependency mismatches in the community.

While solutions to the dependency mismatches are not the focus of our contribution, here we discuss some possibilities:

Stable tracepoints. The kernel developers can establish and maintain a set of stable tracepoints. eBPF developers using these tracepoints can be confident that their programs will continue to work with new kernels. This topic has been discussed in the kernel community for years [34–39], but no concrete proposal has been made.

Compatibility layer. The eBPF community can collectively develop a compatibility layer to provide a stable API over the changing kernel internals. This would help eBPF programs remain functional across kernels while reducing maintenance burden on individual developers. A similar effort has been made for probes in DTrace [20, 79].

Technical Challenges. Some of the challenges we identified are technical and can be addressed through improved tooling and infrastructure.

Function inline. Compilers emit a debug entry for every inlined function, containing information about its name, address, and parameter locations [31, 32]. We leverage this data to implement a proof-of-concept solution [182] that enables tracing of inlined functions. It identifies all inlined call sites of a target function and attaches kprobes both to these inlined locations (within the caller's body) and to the original function. Since inlined functions do not follow standard calling conventions, special handling is needed to correctly access function arguments at each call site. The kernel community are also exploring to encode call-site information into the BTF Type Format (BTF) to better handle these inline cases with CO-RE [109, 150].

Function transformation. Compilers currently provide only basic information about function transformations through name suffixes. We suggest that compilers should emit more comprehensive debug information detailing how functions are transformed. This would enable the eBPF infrastructure to better understand and handle these transformations. While there have been initial efforts to encode transformed functions without parameter changes in BTF [110], full support requires additional information from the compiler.

Functions with the same name. Prior to Linux v6.6 (b022f0c), when there are multiple functions with the same name, an eBPF program is attached to the first one found in the symbol table, which are likely not the intention. The commit changed this behavior to return an error [161]. However, handling the error case effectively requires additional information from the kernel. Several patches have been proposed to address this issue by disambiguating functions using the module and filename [5], a random number [22], or the filename and line number [23], but none have been accepted.

Type checking. To avoid stray reads, we suggest having a standardized mechanism for eBPF programs to explicitly declare the expected type of kernel constructs. This would allow better type checking and avoid silent incorrect results. BTF-enabled eBPF programs represent one step in this direction by tracking and inferring the types on the kernel side [40, 118, 152]. However, this does not work for kprobes, which rely on untyped register values read from `struct pt_regs` and untyped pointer passed to `bpf_probe_read`.

Limitations. DepSurf does not understand the semantics of kernel constructs. Supporting semantic differences is notably challenging considering the kernel's vast codebase and complexity. In addition, our study is comprehensive but not guaranteed to be complete. Covering all combinations that can possibly change the dependency surfaces is impractical. Finally, the data collection of DepSurf relies on the relevant information being accurately exposed in the kernel images and eBPF programs. We believe that DepSurf can encourage the standardization of exposing this data, which is orthogonal to our work.

7 Related Work

eBPF Portability. CO-RE (Compile Once, Run Everywhere) enables compiled eBPF programs to run across different kernels [72, 118, 119]. Naively running a compiled eBPF program on a different kernel does not work because the layout of kernel data structures used by the eBPF program can be different. CO-RE addresses this by generating relocation records during compilation of eBPF programs to track which struct fields are accessed [56]. At load time, CO-RE queries the running kernel's type information in BPF Type Format (BTF) [59] to determine the actual field offsets in that kernel version. It then automatically adjusts the program's field accesses to use the correct offsets for the target kernel. However, relocation does not work if recompiling the source code would have failed on the target kernel (e.g., struct missing).

The eBPF instruction set architecture (ISA) [159] and the format of object files [158] are also being standardized. Our work is complementary because the vast number of kernel constructs continue to evolve and are unlikely to be stable.

eBPF Applications. Researchers have explored the use of eBPF across diverse domains: optimizing performance for storage systems [12, 177, 183], distributed protocols [8, 173, 184, 185], server softwares [15, 69], and cloud-native platforms [136]; supporting in-network and on-device packet processing and monitoring [7, 13, 14, 113, 145]; custumizing kernel behavior [21, 89, 92, 166, 186, 187]. We believe that DepSurf will be even more beneficial as the usage scenarios of eBPF increase.

eBPF Enhancements. The reliability of eBPF programs has been explored from various aspects: the verifier [11, 71, 83, 116, 141, 148, 154, 155, 169], the just-in-time compilers [91, 122, 175], helper functions [88], and security [90, 102, 103, 107]. Recent works have optimized eBPF performance [98, 111] and expanded its capabilities [47]. Our work, instead, focuses on the dependency issues for valid eBPF programs.

Software Dependency. The dependency mismatches of normal user programs on the OS kernel have been studied [6, 100, 123, 165], focusing on POSIX API usage patterns and compatibility. General software dependencies have also been studied in other contexts, such as kernel driver [25], software failures [24], upgrade failures [180], C/C++ libraries [99, 130, 153], and Java projects [171]. eBPF programs suffer from new problems due to the broader dependency surfaces.

Linux Evolution. Prior studies have examined various aspects of Linux kernel evolution, such as codebase growth [96], software complexity [86], build system [117], memory management [82], and performance [137]. DepSurf, to the best of our knowledge, is the first to analyze how the changed constructs affect eBPF programs.

8 Conclusion

We present DepSurf, a tool that analyzes kernel images and eBPF programs to diagnose dependency mismatches. DepSurf enables the first comprehensive study of the eBPF dependency issues by analyzing a range of kernel images. We advocate integrating DepSurf into the eBPF program development and maintenance process to improve the portability of eBPF programs across different kernels.

Acknowledgments

We thank our shepherd, Roberto Natella, and the anonymous reviewers of EuroSys '25 for their valuable feedback and comments. We are grateful to kernel maintainer Yonghong Song for insightful discussion. This work was supported by the NSF under award number CNS-2402859. We thank NetApp, Microsoft, InfluxData, and GE HealthCare for their generous support. Jing Liu was partially supported by a Meta PhD Fellowship. We thank the taxpayers of Wisconsin and the federal government for their support of this work. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and may not reflect the views of NSF or any other institutions.

References

[1] sched_ext schedulers and tools. https://github.com/sched-ext/scx.

[2] System V ABI. System V Application Binary Interface AMD64 Architecture Processor Supplement. https://refspecs.linuxbase.org/elf/x86_64-abi-0.99.pdf.

[3] System V ABI. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification. https://refspecs.linuxbase.org/elf/elf.pdf.

[4] Ethan Afantenos. Hash ordering and Hyrum's Law. https://eaftan.github.io/hash-ordering/, 2021.

[5] Nick Alcock. [PATCH modules-next v10 00/13] kallsyms: reliable symbol->address lookup with /proc/kallmodsyms. https://lore.kernel.org/lkml/20221205163157.269335-1-nick.alcock@oracle.com/, 2022.

[6] Vaggelis Atlidakis, Jeremy Andrus, Roxana Geambasu, Dimitris Mitropoulos, and Jason Nieh. Posix abstractions in modern operating systems: The old, the new, and the missing. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–17, 2016.

[7] Maximilian Bachl, Joachim Fabini, and Tanja Zseby. A flow-based ids using machine learning in ebpf. *arXiv preprint arXiv:2102.09980*, 2021.

[8] Joshua Bardinelli, Yifan Zhang, Jianchang Su, Linpu Huang, Aidan Parilla, Rachel Jarvi, Sameer G Kulkarni, and Wei Zhang. hydns: Acceleration of dns through kernel space resolution. In *Proceedings of the ACM SIGCOMM 2024 Workshop on eBPF and Kernel Extensions*, pages 58–64, 2024.

[9] Abenezer Belachew. Hyrum's Law in Golang. https://abenezer.org/blog/hyrum-law-in-golang, 2021.

[10] Brian N Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility safety and performance in the spin operating system. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 267–283, 1995.

[11] Sanjit Bhat and Hovav Shacham. Formal verification of the linux kernel ebpf verifier range analysis, 2022.

[12] Ashish Bijlani and Umakishore Ramachandran. Extension framework for file systems in user space. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 121–134, 2019.

[13] Marco Bonola, Giacomo Belocchi, Angelo Tulumello, Marco Spaziani Brunella, Giuseppe Siracusano, Giuseppe Bianchi, and Roberto Bifulco. Faster software packet processing on {FPGA}{NICs} with {eBPF} program warping. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 987–1004, 2022.

[14] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. hxdp: Efficient software packet processing on fpga nics. *Communications of the ACM*, 65(8):92–100, 2022.

[15] Matthew Butrovich, Karthik Ramanathan, John Rollinson, Wan Shen Lim, William Zhang, Justine Sherry, and Andrew Pavlo. Tigger: A database proxy that bounces with user-bypass. *Proceedings of the VLDB Endowment*, 16(11):3335–3348, 2023.

[16] "Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler". EXE: Automatically Generating Inputs of Death. CCS '06', 2006.

[17] David Calavera and Lorenzo Fontana. *Linux Observability with BPF: Advanced Programming for Performance Analysis and Networking*. O'Reilly Media, 2019.

[18] Calico et al. Calico. https://www.projectcalico.org/.

[19] David Callahan, Keith D Cooper, Ken Kennedy, and Linda Torczon. Interprocedural constant propagation. *ACM SIGPLAN Notices*, 21(7):152–161, 1986.

[20] Bryan Cantrill, Michael W Shapiro, Adam H Leventhal, et al. Dynamic instrumentation of production systems. In *USENIX Annual Technical Conference, General Track*, pages 15–28, 2004.

[21] Xuechun Cao, Shaurya Patel, Soo Yee Lim, Xueyuan Han, and Thomas Pasquier. {FetchBPF}: Customizable prefetching policies in linux with {eBPF}. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 369–378, 2024.

[22] Alessandro Carminati. [PATCH v2] scripts/link-vmlinux.sh: Add alias to duplicate symbols for kallsyms. https://lore.kernel.org/all/20230714150326.1152359-1-alessandro.carminati@gmail.com/T/, 2023.

[23] Alessandro Carminati. [PATCH v3] scripts/link-vmlinux.sh: Add alias to duplicate symbols for kallsyms. https://lore.kernel.org/lkml/20230828080423.3539686-1-alessandro.carminati@gmail.com/T/#u, 2023.

[24] Marcelo Cataldo, Audris Mockus, Jeffrey A Roberts, and James D Herbsleb. Software dependencies, work dependencies, and their impact on failures. *IEEE Transactions on Software Engineering*, 35(6):864–878, 2009.

[25] Weiteng Chen, Yu Hao, Zheng Zhang, Xiaochen Zou, Dhilung Kirat, Shachee Mishra, Douglas Schales, Jiyong Jang, and Zhiyun Qian. Syzgen++: Dependency inference for augmenting kernel driver fuzzing. In *IEEE Symposium on Security and Privacy*, 2024.

[26] Cilium et al. Cilium. https://cilium.io/.

[27] Cilium et al. Tetragon: eBPF-based Security Observability and Runtime Enforcement. https://github.com/cilium/tetragon.

[28] Cloudflare. eBPF at Cloudflare. https://blog.cloudflare.com/tag/ebpf/.

[29] Cloudflare. Live-patching security vulnerabilities inside the Linux kernel with eBPF Linux Security Module. https://blog.cloudflare.com/live-patch-security-vulnerabilities-with-ebpf-lsm.

[30] cobrien7. libbpf-tools: fix kernel version checks. https://github.com/iovisor/bcc/commit/312a40de6d72d9969999aed991ae066d383639bb, 2022.

[31] DWARF Standard Committee. DWARF Debugging Information Format Version 4. https://dwarfstd.org/doc/DWARF4.pdf.

[32] DWARF Standard Committee. DWARF Debugging Information Format Version 5. https://dwarfstd.org/doc/DWARF5.pdf.

[33] Jonathan Corbet. Maintainers Summit topics: pull depth, hardware vulnerabilities, etc. https://lwn.net/Articles/799262/.

[34] Jonathan Corbet. ABI status for tracepoints. https://lwn.net/Articles/412685/, 2010.

[35] Jonathan Corbet. Statistics and tracepoints. https://lwn.net/Articles/401769/, 2010.

[36] Jonathan Corbet. Two ABI troubles. https://lwn.net/Articles/408689/, 2010.

[37] Jonathan Corbet. Ftrace, perf, and the tracing ABI. https://lwn.net/Articles/442113/, 2011.

[38] Jonathan Corbet. Another attempt to address the tracepoint ABI problem. https://lwn.net/Articles/737530/, 2017.

[39] Jonathan Corbet. Dynamic function tracing events. https://lwn.net/Articles/747256/, 2018.

[40] Jonathan Corbet. Type checking for BPF tracing. https://lwn.net/Articles/803258/, 2019.

[41] Jonathan Corbet. Git archive generation meets Hyrum's law. https://lwn.net/Articles/921787/, 2023.

[42] Alban Crequy. Using BPF Iterators to Gain Insight into Kubernetes. https://www.youtube.com/watch?v=ilcYXPDSgu8&list=PLj6h78yzYM2Pm5nF_GmNQHMyt9CUZr2uQ&index=6.

[43] Deepfence. eBPFGuard: Rust library for writing Linux security policies using eBPF. https://github.com/deepfence/ebpfguard.

[44] Deepfence. Introducing eBPFGuard: A Library for Inline Mitigation of Threats using LSM Hooks. https://www.deepfence.io/blog/ebpfguard-a-library-for-inline-mitigation-of-threats.

[45] Mugdha Deokar, Jingyang Men, Lucas Castanheira, Ayush Bhardwaj, and Theophilus A. Benson. An empirical study on the challenges of ebpf application development. In *Proceedings of the ACM SIGCOMM 2024 Workshop on EBPF and Kernel Extensions*, eBPF '24, page 1–8,

New York, NY, USA, 2024. Association for Computing Machinery.
[46] Peter Druschel, Vivek S Pai, and Willy Zwaenepoel. Extensible kernels are leading os research astray. HOTOS '97, page 38, USA, 1997. IEEE Computer Society.
[47] Kumar Kartikeya Dwivedi, Rishabh Iyer, and Sanidhya Kashyap. Fast, flexible, and practical kernel extensions. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 249–264, 2024.
[48] Dawson R Engler, M Frans Kaashoek, and James O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. *ACM SIGOPS Operating Systems Review*, 29(5):251–266, 1995.
[49] Facebook. Katran: High-Performance Layer 4 Load Balancing. https://github.com/facebookincubator/katran.
[50] Facebook. Open-sourcing Katran: a scalable network load balancer. https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer/.
[51] Stanislav Fomichev, Eric Dumazet, , Willem de Bruijn, Vlad Dumitrescu, Bill Sommerfeld, and Peter Oskolkov. Replacing HTB with EDT and BPF. https://netdevconf.info//0x14/session.html?talk-replacing-HTB-with-EDT-and-BPF.
[52] Linux Foundation. Backporting and conflict resolution. https://www.kernel.org/doc/html/latest/process/backporting.html.
[53] Linux Foundation. BPF CO-RE (Compile Once - Run Everywhere). https://docs.kernel.org/bpf/libbpf/libbpf_overview.html#bpf-co-re-compile-once-run-everywhere.
[54] Linux Foundation. BPF Documentation. https://docs.kernel.org/bpf/.
[55] Linux Foundation. BPF Kernel Functions. https://docs.kernel.org/bpf/kfuncs.html.
[56] Linux Foundation. BPF LLVM Relocations. https://docs.kernel.org/bpf/llvm_reloc.html.
[57] Linux Foundation. BPF LSM. https://docs.kernel.org/bpf/prog_lsm.html.
[58] Linux Foundation. BPF Program Types. https://docs.kernel.org/bpf/libbpf/program_types.html.
[59] Linux Foundation. BPF Type Format (BTF). https://www.kernel.org/doc/html/next/bpf/btf.html.
[60] Linux Foundation. Kprobes. https://docs.kernel.org/trace/kprobes.html.
[61] Linux Foundation. Linux Security Modules. https://docs.kernel.org/security/lsm.html.
[62] Linux Foundation. Tracepoints. https://docs.kernel.org/trace/tracepoints.html.
[63] Linux Foundation. Uprobe-tracer: Uprobe-based Event Tracing. https://docs.kernel.org/trace/uprobetracer.html.
[64] Iago López Galeiras. Add RestrictFileSystems= property using LSM BPF. https://github.com/systemd/systemd/pull/18145.
[65] Bolaji Gbadamosi, Luigi Leonardi, Tobias Pulls, Toke Høiland-Jørgensen, Simone Ferlin-Reiter, Simo Sorce, and Anna Brunström. The ebpf runtime in the linux kernel, 2024.
[66] GCC. GCC Options That Control Optimization. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.
[67] GCC. Interprocedural Optimization Passes. https://gcc.gnu.org/onlinedocs/gccint/IPA-passes.html.
[68] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly 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*, pages 1069–1084, 2019.
[69] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. BMC: Accelerating memcached using safe in-kernel caching and pre-stack processing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 487–501, 2021.
[70] GitHub. Debugging network stalls on Kubernetes. https://github.blog/2019-11-21-debugging-network-stalls-on-kubernetes/.
[71] Google. Buzzer - An eBPF Fuzzer toolchain. https://github.com/google/buzzer.
[72] Brendan Gregg. BPF binaries: BTF, CO-RE, and the future of BPF perf tools. https://www.brendangregg.com/blog/2020-11-04/bpf-co-re-btf-libbpf.html.
[73] Brendan Gregg. Linux BPF Superpowers. https://www.brendangregg.com/blog/2016-03-05/linux-bpf-superpowers.html.
[74] Brendan Gregg. *BPF Performance Tools*. Addison-Wesley Professional, 2019. http://www.brendangregg.com/bpf-performance-tools-book.html.
[75] Brendan Gregg et al. BCC. https://github.com/iovisor/bcc.
[76] Brendan Gregg et al. biotop. https://github.com/iovisor/bcc/blob/master/libbpf-tools/biotop.bpf.c.
[77] Brendan Gregg et al. readahead. https://github.com/iovisor/bcc/blob/master/libbpf-tools/readahead.bpf.c.
[78] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Newman Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. {CertiKOS}: An extensible architecture for building certified concurrent {OS} kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 653–669, 2016.
[79] Kris Van Hees. Tooling for semantic probing based on BPF and kernel tracing. https://lpc.events/event/18/contributions/1925/, 2024.
[80] Ism Hong. Incorrect result while running biolatency.py with flags option on kernel 4.9.266. https://github.com/iovisor/bcc/issues/3587.
[81] Ism Hong. runqlen / cpuunclaimed get wrong result for Linux kernel > 5.7.0. https://github.com/iovisor/bcc/issues/4602.
[82] Jian Huang, Moinuddin K Qureshi, and Karsten Schwan. An evolutionary study of linux memory management for fun and profit. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 465–478, 2016.
[83] Hsin-Wei Hung and Ardalan Amiri Sani. Brf: ebpf runtime fuzzer. *arXiv preprint arXiv:2305.08782*, 2023.
[84] hygoni. mm/slab common: unify NUMA and UMA version of tracepoints. https://github.com/torvalds/linux/commit/11e9734bcb6a7361943f993eba4e97f5812120d8.
[85] IETF BPF Working Group. eBPF ELF Profile Specification, v0.1. https://datatracker.ietf.org/doc/html/draft-thaler-bpf-elf.
[86] Ayelet Israeli and Dror G Feitelson. The linux kernel as a case study in software evolution. *Journal of Systems and Software*, 83(3):485–501, 2010.
[87] Jinghao Jia, Raj Sahu, Adam Oswald, Dan Williams, Michael V Le, and Tianyin Xu. Kernel extension verification is untenable. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, pages 150–157, 2023.
[88] Jinghao Jia, Raj Sahu, Adam Oswald, Dan Williams, Michael V Le, and Tianyin Xu. Kernel extension verification is untenable. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, pages 150–157, 2023.
[89] Jinghao Jia, YiFei Zhu, Dan Williams, Andrea Arcangeli, Claudio Canella, Hubertus Franke, Tobin Feldman-Fitzthum, Dimitrios Skarlatos, Daniel Gruss, and Tianyin Xu. Programmable system call security with ebpf. *arXiv preprint arXiv:2302.10366*, 2023.
[90] Di Jin, Vaggelis Atlidakis, and Vasileios P Kemerlis. Epf: Evil packet filter. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 735–751, 2023.
[91] Di Jin, Alexander J Gaidis, and Vasileios P Kemerlis. {BeeBox}: Hardening {BPF} against transient execution attacks. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 613–630, 2024.
[92] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. Syrup: User-defined scheduling across the stack. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 605–620, 2021.

[93] Zen Kernel. Zen Patched Kernel Sources. https://github.com/zen-kernel/zen-kernel.

[94] Kinvolk. Extending systemd security features with eBPF. https://kinvolk.io/blog/2021/04/extending-systemd-security-features-with-ebpf/.

[95] Tomasz Knopik. cachestat produces incorrect hit ratio. https://github.com/iovisor/bcc/issues/2366.

[96] Oded Koren. A study of the linux kernel evolution. *ACM SIGOPS Operating Systems Review*, 40(2):110–112, 2006.

[97] KubeArmor et al. KubeArmor. https://kubearmor.io/.

[98] Hsuan-Chi Kuo, Kai-Hsun Chen, Yicheng Lu, Dan Williams, Sibin Mohan, and Tianyin Xu. Verified programs can party: optimizing kernel extensions via post-verification merging. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 283–299, 2022.

[99] Gleb Kurtsou. shlib-compat - ABI compatibility checker for shared libraries with symbol versioning. https://github.com/glk/shlib-compat.

[100] Hugo Lefeuvre, Gaulthier Gain, Vlad-Andrei Bădoiu, Daniel Dinca, Vlad-Radu Schiller, Costin Raiciu, Felipe Huici, and Pierre Olivier. Loupe: Driving the development of os compatibility layers. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 249–267, 2024.

[101] Zhichuan Liang. Real World Debugging with eBPF. https://www.usenix.org/system/files/srecon23apac_slides-liang_zhichuan.pdf.

[102] Soo Yee Lim, Xueyuan Han, and Thomas Pasquier. Unleashing unprivileged ebpf potential with dynamic sandboxing. In *Proceedings of the 1st Workshop on eBPF and Kernel Extensions*, pages 42–48, 2023.

[103] Soo Yee Lim, Tanya Prasad, Xueyuan Han, and Thomas Pasquier. Safebpf: Hardware-assisted defense-in-depth for ebpf kernel extensions. In *Proceedings of the 2024 on Cloud Computing Security Workshop*, pages 80–94, 2024.

[104] Arch Linux. Realtime Kernel Patchset. https://wiki.archlinux.org/title/Realtime_kernel_patchset.

[105] Linux Lock et al. bpflock - eBPF driven security for locking and auditing Linux machines. https://github.com/linux-lock/bpflock.

[106] Lockc et al. Lockc: Making containers more secure with eBPF and Linux Security Modules (LSM). https://github.com/lockc-project/lockc.

[107] Hongyi Lu, Shuai Wang, Yechang Wu, Wanning He, and Fengwei Zhang. Moat: Towards safe bpf kernel extension. *arXiv preprint arXiv:2301.13421*, 2023.

[108] Lumontec. Some freshness with Linux security modules and eBPF. https://medium.com/@lumontec/some-freshness-with-linux-security-modules-and-ebpf-676ac363a135.

[109] Alan Maguire. More co-re? taming the effects of compiler optimizations on bpf tracing. https://lpc.events/event/16/contributions/1371/, 2023.

[110] Alan Maguire. [patch v3 dwarves 0/8] dwarves: support encoding of optimized-out parameters, removal of inconsistent static functions. https://lore.kernel.org/bpf/1675790102-23037-1-git-send-email-alan.maguire@oracle.com/, 2023.

[111] Jinsong Mao, Hailun Ding, Juan Zhai, and Shiqing Ma. Merlin: Multi-tier Optimization of eBPF Code for Performance and Compactness. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 639–653, 2024.

[112] Jerome Marchand. tools: Add support for the new block_io_* tracepoints. https://github.com/iovisor/bcc/pull/4691.

[113] Sebastiano Miano, Xiaoqi Chen, Ran Ben Basat, and Gianni Antichi. Fast in-kernel traffic sketching in ebpf. *ACM SIGCOMM Computer Communication Review*, 53(1):3–13, 2023.

[114] michael chuh. tools/ttysnoop: Fix wrong KERNEL_VERSION. https://github.com/iovisor/bcc/commit/4fe6a6ae038eb3a70460be6619d697ffd8531738, 2022.

[115] Samantha Miller, Kaiyuan Zhang, Danyang Zhuo, Shibin Xu, Arvind Krishnamurthy, and Thomas Anderson. Practical safe linux kernel extensibility. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 170–176, 2019.

[116] Mohamed Husain Noor Mohamed, Xiaoguang Wang, and Binoy Ravindran. Understanding the security of linux ebpf subsystem. In *Proceedings of the 14th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 87–92, 2023.

[117] Sarah Nadi and Ric Holt. The linux kernel: A case study of build system variability. *Journal of Software: Evolution and Process*, 26(8):730–746, 2014.

[118] Andrii Nakryiko. BPF Core Reference Guide. https://nakryiko.com/posts/bpf-core-reference-guide/.

[119] Andrii Nakryiko. BPF Portability and Co-RE. https://nakryiko.com/posts/bpf-portability-and-co-re/.

[120] Andrii Nakryiko et al. libbpf. https://github.com/libbpf/libbpf.

[121] George C Necula and Peter Lee. Safe kernel extensions without run-time checking. In *OSDI*, volume 96, pages 229–243, 1996.

[122] Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang. Specification and verification in the field: Applying formal methods to BPF just-in-time compilers in the linux kernel. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 41–61, 2020.

[123] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. A Binary-compatible Unikernel. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 59–73, 2019.

[124] OpenLogic. Top Enterprise Linux Distributions. https://www.openlogic.com/blog/top-enterprise-linux-distributions.

[125] Linux Manual Pages. BPF Helpers. https://man7.org/linux/man-pages/man7/bpf-helpers.7.html.

[126] Linux Manual Pages. tc-bpf(8) — Linux manual page. https://man7.org/linux/man-pages/man8/tc-bpf.8.html.

[127] Siddhartha Patil. [PATCH 0/1] Revert change in pipe reader wakeup behavior. https://lore.kernel.org/lkml/20210729222635.2937453-1-sspatil@android.com/T/#u, 2021.

[128] Pixie et al. Pixie: Instant Kubernetes-Native Application Observability. https://github.com/pixie-io/pixie.

[129] Tanel Poder. When eBPF task->stack->pt_regs reads return garbage on the latest Linux kernels, blame Fred! https://tanelpoder.com/posts/ebpf-pt-regs-error-on-linux-blame-fred/, 2025.

[130] Andrey Ponomarenko. ABI Compliance Checker. https://lvc.github.io/abi-compliance-checker/.

[131] Elvis Pranskevichus and Yury Selivanov. What's New In Python 3.6. https://docs.python.org/3.6/whatsnew/3.6.html#new-dict-implementation, 2016.

[132] BCC Project. biotop and biosnoop do not work under 5.19 kernel due to missing blk_account_io_start kprobe. https://github.com/iovisor/bcc/issues/4261.

[133] BCC Project. block tracepoints no longer have struct request queue arg. https://github.com/iovisor/bcc/commit/3766f48c94372698b71c8c46ab1142c8f6dab9f0.

[134] BCC Project. libbpf-tools: fix for block io tracepoints changed. https://github.com/iovisor/bcc/commit/3766f48c94372698b71c8c46ab1142c8f6dab9f0.

[135] IO Visor Project. XDP: eXpress Data Path. https://www.iovisor.org/technology/xdp.

[136] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, and KK Ramakrishnan. Spright: extracting the server from serverless computing! high-performance ebpf-based event-driven, shared-memory processing. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 780–794, 2022.

[137] Xiang Ren, Kirk Rodrigues, Luyuan Chen, Camilo Vega, Michael Stumm, and Ding Yuan. An analysis of performance evolution of linux's core operations. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 554–569, 2019.

[138] Alastair Robertson et al. bpftrace. https://github.com/bpftrace/bpftrace.

[139] Steven Rostedt. [Ksummit-discuss] [MAINTAINERS SUMMIT] How far to go with eBPF. https://lore.kernel.org/all/20220616122634.6e11e58c@gandalf.local.home/.

[140] Eric Sage and Melissa Kilby. Think eBPF for Kernel Security Monitoring Falco at Apple Eric Sage & Melissa Kilby, Apple. https://www.youtube.com/watch?v=ZBlJSr6XkN8.

[141] Simon Scannell. Fuzzing for eBPF JIT bugs in the Linux kernel. https://scannell.io/posts/ebpf-fuzzing/.

[142] Casey Schaufler. Re: [PATCH bpf-next v13 4/7] landlock: Add ptrace LSM hooks. https://lore.kernel.org/lkml/637736ef-c48e-ac3b-3eef-8a6a095a96f1@schaufler-ca.com/.

[143] Aqua Security et al. Tracee: Linux Runtime Security and Forensics using eBPF. https://github.com/aquasecurity/tracee.

[144] Margo I Seltzer, Yasuhiro Endo, Christopher Small, and Keith A Smith. Dealing with disaster: Surviving misbehaved kernel extensions. *SIGOPS Operating Systems Review*, 30(213-228):10–1145, 1996.

[145] Junxian Shen, Han Zhang, Yang Xiang, Xingang Shi, Xinrui Li, Yunxi Shen, Zijian Zhang, Yongxiang Wu, Xia Yin, Jilong Wang, et al. Network-centric distributed tracing with deepflow: Troubleshooting your microservices in zero code. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 420–437, 2023.

[146] KP Singh. Re: [PATCH bpf-next v1 00/13] MAC and Audit policy using eBPF (KRSI). https://lwn.net/ml/linux-kernel/CACYkzJ5nYh7eGuru4vQ=2ZWumGPszBRbgqxmhd4WQRXktAUKkQ@mail.gmail.com/.

[147] KP Singh. Security Auditing and Enforcement using eBPF - KP Singh, Google - Full Keynote. https://www.youtube.com/watch?v=XFJw37Vwzcc.

[148] snorez. ebpf-fuzzer: fuzz the linux kernel bpf verifier. https://github.com/snorez/ebpf-fuzzer.

[149] Yonghong Song. [PATCH bpf-next] bcc: Fix ttysnoop for kernel > v5.11 and other config changes. https://github.com/iovisor/bcc/pull/3360/files#r611360515, 2021.

[150] Yonghong Song and Alan Maguire. Kernel func tracing in the face of compiler optimization. https://lpc.events/event/18/contributions/1945/, 2024.

[151] Alexei Starovoitov. Re: [PATCH bpf-next v13 4/7] landlock: Add ptrace LSM hooks. https://lore.kernel.org/lkml/20191105215453.szhdkrvuekwfz6le@ast-mbp.dhcp.thefacebook.com/.

[152] Alexei Starovoitov. [PATCH v3 bpf-next 00/11] bpf: revolutionize bpf tracing. https://https://lore.kernel.org/all/20191016032505.2089704-1-ast@kernel.org/, 2019.

[153] Andrew Suffield. icheck - C interface ABI/API checker. https://manpages.ubuntu.com/manpages/noble/man1/icheck.1.html.

[154] Hao Sun and Zhendong Su. Validating the {eBPF} verifier via state embedding. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 615–628, 2024.

[155] Hao Sun, Yiru Xu, Jianzhong Liu, Yuheng Shen, Nan Guan, and Yu Jiang. Finding correctness bugs in ebpf verifier with structured and sanitized program. 2024.

[156] Sysdig et al. Falco: Cloud Native Runtime Security. https://github.com/falcosecurity/falco.

[157] systemd. systemd RestrictFileSystems=. https://www.freedesktop.org/software/systemd/man/latest/systemd.exec.html#RestrictFileSystems=.

[158] Dave Thaler. eBPF ELF Profile Specification, v0.1. Internet-Draft draft-thaler-bpf-elf-00, Internet Engineering Task Force, March 2023. Work in Progress.

[159] Dave Thaler. BPF Instruction Set Architecture (ISA). Internet-Draft draft-ietf-bpf-isa-03, Internet Engineering Task Force, May 2024. Work in Progress.

[160] Alok Tiagi, Hariharan Ananthakrishnan, Ivan Porto Carrero, and Keerti Lakshminarayan. How Netflix uses eBPF flow logs at scale for network insight. https://netflixtechblog.com/how-netflix-uses-ebpf-flow-logs-at-scale-for-network-insight-e3ea997dca96.

[161] Rafael David Tinoco. Cannot create probe on symbols with duplicate entries in kallsyms (multiple addresses) in recent kernels. https://github.com/aquasecurity/tracee/issues/3653, 2023.

[162] Linus Torvalds. Re: [BREAKAGE] Since 4.18, kernel sets SB_I_NODEV implicitly on userns mounts, breaking systemd-nspawn. https://lkml.org/lkml/2018/12/22/232.

[163] Linus Torvalds. Re: LVM snapshot broke between 4.14 and 4.16. https://lkml.org/lkml/2018/8/3/621.

[164] Linus Torvalds. Re: [Regression w/ patch] Media commit causes user space to misbahave (was: Re: Linux 3.8-rc1). https://lkml.org/lkml/2012/12/23/75.

[165] Chia-Che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald E Porter. A study of modern linux api usage and compatibility: What to support when you're supporting. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–16, 2016.

[166] Manas Tungare, Pardha S Pyla, Miten Sampat, and Manuel A Pérez-Quinones. Syncables: A framework to support seamless data migration across multiple platforms. In *2007 IEEE International Conference on Portable Information Devices*, pages 1–5. IEEE, 2007.

[167] Ubuntu. Ubuntu Linux Packages. https://launchpad.net/ubuntu/+source/linux/.

[168] Arjan van de Ven. Fix powerTOP regression with 2.6.39-rc5. https://lore.kernel.org/all/4DC45537.6070609@linux.intel.com/T/#u, 2011.

[169] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. Verifying the verifier: ebpf range analysis verification. In *International Conference on Computer Aided Verification*, pages 226–251. Springer, 2023.

[170] Krister Walfridsson. Interprocedural Optimization in GCC. https://kristerw.blogspot.com/2017/05/interprocedural-optimization-in-gcc.html.

[171] Ying Wang, Ming Wen, Zhenwei Liu, Rongxin Wu, Rui Wang, Bo Yang, Hai Yu, Zhiliang Zhu, and Shing-Chi Cheung. Do the dependency conflicts in my project matter? In *Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 319–330, 2018.

[172] Matthew Wilcox. [PATCH v14 000/138] Memory folios. https://lore.kernel.org/all/20210715033704.692967-1-willy@infradead.org/T/#u.

[173] Thomas Wirtgen, Tom Rousseaux, Quentin De Coninck, Nicolas Rybowski, Randy Bush, Laurent Vanbever, Axel Legay, and Olivier Bonaventure. {xBGP}: Faster innovation in routing protocols. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 575–592, 2023.

[174] Hyrum Wright. Hyrum's law. https://www.hyrumslaw.com/.

[175] Xi Wang and David Lazar and Nickolai Zeldovich and Adam Chlipala and Zachary Tatlock. Jitk: A trustworthy In-Kernel interpreter infrastructure. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 33–47, Broomfield, CO, October 2014. USENIX Association.

[176] Xiheng. How Does Alibaba Cloud Build High-Performance Cloud-Native Pod Networks in Production Environments? https://www.alibabacloud.com/blog/how-does-alibaba-cloud-build-high-performance-cloud-native-pod-networks-in-production-environments_596590.

[177] Zhe Yang, Youyou Lu, Xiaojian Liao, Youmin Chen, Junru Li, Siyu He, and Jiwu Shu. λ-IO: A unified IO stack for computational storage. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*,

pages 347–362, 2023.

[178] Hyeonggon Yoo. mm/slab common: unify NUMA and UMA version of tracepoints. https://github.com/torvalds/linux/commit/11e9734.

[179] ZDNet. Linus Torvalds talks about coming back to work on Linux. https://www.zdnet.com/article/linus-torvalds-talks-about-coming-back-to-work-on-linux/.

[180] Yongle Zhang, Junwen Yang, Zhuqi Jin, Utsav Sethi, Kirk Rodrigues, Shan Lu, and Ding Yuan. Understanding and detecting software upgrade failures in distributed systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 116–131, 2021.

[181] Shawn Zhong. libpf-tools/readahead: Fix attachment failure since v5.16. https://github.com/iovisor/bcc/pull/5086, 2024.

[182] Shawn Zhong. [proposal] tracing inline functions. https://github.com/iovisor/bcc/issues/5093, 2024.

[183] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, et al. XRP:In-Kernel storage functions with eBPF. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 375–393, 2022.

[184] Yang Zhou, Zezhou Wang, Sowmya Dharanipragada, and Minlan Yu. Electrode: Accelerating distributed protocols with eBPF. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1391–1407, 2023.

[185] Yang Zhou, Xingyu Xiang, Matthew Kiley, Sowmya Dharanipragada, and Minlan Yu. {DINT}: Fast {In-Kernel} distributed transactions with {eBPF}. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 401–417, 2024.

[186] Tal Zussman, Teng Jiang, and Asaf Cidon. Custom page fault handling with ebpf. In *Proceedings of the ACM SIGCOMM 2024 Workshop on eBPF and Kernel Extensions*, pages 71–73, 2024.

[187] Tal Zussman, Ioannis Zarkadas, Jeremy Carin, Andrew Cheng, Hubertus Franke, Jonas Pfefferle, and Asaf Cidon. Cache is king: Smart page eviction with ebpf. *arXiv preprint arXiv:2502.02750*, 2025.

A Artifact Appendix

A.1 Abstract

The artifact for DepSurf includes the following components:

- The source code of the `depsurf` Python library that extracts and analyzes kernel dependency surfaces, and generates dependency reports for eBPF programs.
- A set of Jupyter notebooks to drive the `depsurf` library and reproduce the experiments in the paper.
- A dataset of kernel dependency surfaces, including information about the kernel functions, structs, tracepoints, and system calls.

A.2 Description & Requirements

A.2.1 How to access

The artifact for this paper is available at https://github.com/ShawnZhong/DepSurf in branch `eurosys25` and archived at https://doi.org/10.5281/zenodo.14881217.

A.2.2 Hardware dependencies

Generating the dataset requires a machine with at least 64GB of disk space to store the intermediate data produced.

A.2.3 Software dependencies

The artifact requires Python 3.11 or higher with the following packages:

- `launchpadlib>=2.1.0` (optional)
- `matplotlib>=3.10.0`
- `notebook>=7.3.2`
- `pandas>=2.2.3`
- `pyelftools>=0.31`

Additional dependencies are listed in 00_deps.ipynb.

A.2.4 Benchmarks

The dataset is available at https://github.com/ShawnZhong/DepSurf-dataset. The code used to generate it is provided in the main repository. Here we provide some examples of the format of the dataset.

Function Status. The following shows the status of `vfs_fsync`.

```
{
  "name": "vfs_fsync",
  "collision_type": "Unique Global",
  "inline_type": "Partially inlined",
  "funcs": [
    {
      "addr": 18446744071582330835,
      "name": "vfs_fsync",
      "external": true,
      "loc": "fs/sync.c:213",
      "file": "fs/sync.c",
      "inline": "not declared, inlined",
      "caller_inline": [
        "fs/sync.c:__ia32_sys_fdatasync",
        "fs/sync.c:__x64_sys_fdatasync",
        "fs/sync.c:__ia32_sys_fsync",
        "fs/sync.c:__x64_sys_fsync"
      ],
      "caller_func": [
        "fs/aio.c:aio_fsync_work",
        "fs/iomap/swapfile.c:iomap_swapfile_activate",
        "drivers/block/loop.c:do_req_filebacked",
        "drivers/block/loop.c:__loop_update_dio",
        "drivers/md/md-bitmap.c:md_bitmap_create"
      ]
    }
  ],
  "symbols": [
    {
      "addr": 18446744071582330944,
      "name": "vfs_fsync",
      "section": ".text",
      "bind": "STB_GLOBAL",
      "size": 121
    }
  ]
}
```

Function Declaration. The following shows the type of `vfs_fsync`.

```
{
  "kind": "FUNC",
  "name": "vfs_fsync",
  "type": {
    "kind": "FUNC_PROTO",
    "params": [
      {
        "name": "file",
        "type": {
          "kind": "PTR",
          "type": {
            "kind": "STRUCT",
            "name": "file"
          }
        }
      },
      {
        "name": "datasync",
        "type": {
          "kind": "INT",
          "name": "int"
        }
      }
    ],
    "ret_type": {
      "kind": "INT",
      "name": "int"
    }
  }
}
```

Struct. The following shows the type of `timespec`.

```
{
  "kind": "STRUCT",
  "name": "timespec",
  "size": 16,
  "members": [
    {
      "name": "tv_sec",
      "bits_offset": 0,
      "type": {
        "kind": "TYPEDEF",
        "name": "__kernel_time_t"
      }
    },
    {
      "name": "tv_nsec",
      "bits_offset": 64,
      "type": {
        "kind": "INT",
        "name": "long int"
      }
    }
  ]
}
```

Tracepoint. The following shows an excerpt of tracepoint `timer_init`.

```
{
  "class_name": "timer_class",
  "event_name": "timer_init",
  "func_name": "trace_event_raw_event_timer_class",
  "struct_name": "trace_event_raw_timer_class",
  "fmt_str": "\"timer=%p\", REC->timer",
  "func": {
    "kind": "FUNC",
    "name": "trace_event_raw_event_timer_class",
    "type": {
      "kind": "FUNC_PROTO",
      "params": [
        {
          "name": "__data",
          "type": {
            "kind": "PTR",
            "type": {
              "name": "void",
              "kind": "VOID"
            }
          }
        },
        {
          "name": "timer",
          "type": {
            "kind": "PTR",
            "type": {
              "kind": "STRUCT",
              "name": "timer_list"
            }
          }
        }
      ],
      "ret_type": {
        "name": "void",
        "kind": "VOID"
      }
    }
  },
  "struct": {
    "kind": "STRUCT",
    "name": "trace_event_raw_timer_class",
    "size": 16,
    "members": [
      {
        "name": "ent",
        "bits_offset": 0,
        "type": {
          "kind": "STRUCT",
          "name": "trace_entry"
        }
      }, // ...
    ]
  }
}
```

A.3 Set-up

Please follow the README.md in the main repository to set up the environment.

A.4 Evaluation workflow

A.4.1 Major Claims

- (C1): Kernel source code evolution significantly impacts dependency surfaces, as demonstrated by experiments (E1) with results in Table 3 and Table 4.
- (C2): Different kernel configurations lead to variations in dependency surfaces, as shown by experiments (E2) with results in Table 5.
- (C3): The kernel build process affects dependency surfaces. This is validated by experiments (E3) with results in Figure 5, Figure 6, and Table 6.
- (C4): Dependency mismatches have a broad impact on eBPF programs, as revealed by experiments (E4) with results in Table 7 and Table 8.

A.4.2 Experiments

- (E1) Analysis on Kernel Source Code [5 minutes]: To reproduce the analysis results on kernel source code evolution, run:
  - 30_diff.ipynb to generate the diffs
  - 35_src.ipynb for Table 3
  - 36_breakdown.ipynb for Table 4
- (E2) Analysis on Kernel Configuration [5 minutes]: To analyze how different kernel configurations affect dependency surfaces, run:
  - 30_diff.ipynb to generate the diffs
  - 39_config.ipynb to reproduce Table 5
- (E3) Analysis on Kernel Compilation [1 minute]: To examine compilation effects on dependency surfaces, run:
  - 40_inline.ipynb for function inlining (Figure 5)
  - 41_transform.ipynb for function transformation (Figure 6)
  - 42_dup.ipynb for function duplication (Table 6)
- (E4) eBPF Program Analysis [5 minutes]: To analyze dependency mismatches in eBPF programs, run:
  - 50_programs.ipynb to compile and analyze eBPF programs
  - 52_summary.ipynb for summary statistics (Table 7 and Table 8)

A.5 Notes on Reusability

Additional kernel images can be added to the dataset by adding the URL to the kernel image to 11_download.ipynb. Additional eBPF programs can be added to the `PROG_PATHS` list in 50_programs.ipynb.