Unsafe nesting in BPF programs

Siddharth Chintamaneni

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science & Applications

Dan Williams, Chair
Dimitrios S Nikolopoulos
Sam H. Noh

December 13, 2024
Blacksburg, Virginia

Keywords: eBPF, Linux, Kernel extensions, Nesting

Unsafe nesting in BPF programs

Siddharth Chintamaneni

(ABSTRACT)

Safe kernel extensions are crucial for adding features like networking filters, security policies, and monitoring capabilities that organizations require in production environments. The Linux kernel traditionally lacked mechanisms for safe runtime extensions. BPF addressed this problem by enabling dynamic kernel extensions with safety guarantees enforced by an in-kernel verifier, ensuring kernel stability. The verifier verifies each BPF program without considering its interactions with other BPF programs, assuming these interactions will be safe. This assumption relies on both static limits enforced by the verifier and runtime checks in the kernel. However, this verification approach leaves the kernel vulnerable to safety issues when BPF programs nest within each other. This work identifies such safety issues, including stack overflows, deadlocks, performance issues, and missed events. To address these challenges, this research presents an approach for providing a global system view to the verifier to prevent uncontrolled nesting. We explored the first steps in this direction through a helper-rooted callgraph approach that provides a global view of BPF program interactions, enabling the prevention of these safety issues.

Unsafe nesting in BPF programs

Siddharth Chintamaneni

(GENERAL AUDIENCE ABSTRACT)

Linux is one of the most popular mainstream operating systems, running on over 96% of the world's servers. Extending the kernel is important because users need features like network filtering and system monitoring. Recently, BPF provided a safe way to extend Linux's capabilities by using an in-kernel safety checker called the verifier, which examines each program before it runs to prevent crashes and system stalls. Due to these safety guarantees, BPF has been widely adopted in industry for various use cases.

In this work, we identified that when multiple BPF programs are nested, they can cause problems even though the programs are individually verified as safe. Our experiments showed three critical issues: stack overflows, deadlocks and performance problems (such as throughput loss). We identified that these crashes occur because the verifier lacks knowledge about how BPF programs interact with each other. To address this problem, we developed an approach called helper-rooted callgraphs that shows how different programs interact. This information can then be used by the verifier to prevent unsafe program interactions.

To my family and friends

Acknowledgments

I would like to express my deepest gratitude to my advisor and mentor, Dr. Dan Williams, whose guidance and motivation were crucial not only for this research but also for shaping my future path. His valuable insights were essential, and without his guidance, this project would not have reached its current state. Along with research guidance, he helped me understand different ways to think about problems, how to abstract high-level ideas, and how to enjoy the process of research and writing. I want to thank my labmates, with whom I spent countless hours over these two years discussing this project and new ideas, and sharing feedback. Special thanks to Egor Lukiyanov, Milo Craun, Raj Sahu, Sai Roop Somaraju, and Zhengjie Ji for helping me along this journey. Lastly, I would like to thank my parents, family members, and friends; without their support and help, I could not have made it this far.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Kernel extensions allow userspace to modify kernel behavior. This ability enables userspace applications to load code into the kernel and accelerate its performance. BPF programs have emerged as the preferred mechanism for safely extending the Linux kernel, gaining widespread industry adoption due to their strong safety guarantees. These guarantees are enforced by an in-kernel static analyzer called the verifier, which ensures BPF programs are safe to execute and won't significantly impact system performance by limiting their instruction count. However, in this thesis, we show that these safety properties can be violated when multiple BPF programs nest within each other. We also show that existing and proposed ad-hoc techniques to prevent these issues have limitations and side effects. To address these challenges, we introduce a callgraph-based static analysis approach that identifies potentially unsafe nesting scenarios and propose a design for integrating this solution into the verifier.

1.1 The safety implications of nested BPF programs

BPF programs are nested when one BPF program triggers the execution of another. During verification, the verifier examines each BPF program as an individual unit and makes assumptions about the runtime environment these programs interact with. These assumptions are based on runtime checks enforced by the BPF subsystem and kernel implementation details. The verifier, through its limits and runtime checks, is intended to ensure that BPF programs, whether running independently or together, cannot crash the kernel. In this work, we provide evidence that the verifier's assumptions are not true.

In Chapter 2, we demonstrate that BPF program nesting is not merely a feature but a requirement for tracing and networking use cases. Chapter 3 presents experimental evidence showing how nested BPF programs can cause serious problems, including stack overflows, deadlocks, and performance issues. Chapter 4 examines current techniques[1] that address some of these issues in the kernel and their limitations. Then we establish that the fundamental problem is the verifier's lack of a global view in Chapter 5.

1.2 Static solution to prevent nesting problems

After identifying the problems with uncontrolled nesting of BPF programs and examining current solutions, we then present our approach to addressing these challenges. Our solution aims to prevent nesting-related issues by providing the verifier with a global view of program interactions before they occur.

To address the limitations of current state-of-the-art techniques, we propose a callgraph-based static solution. This enables the verifier to detect BPF program nesting during veri-

[1]Some of these implementations are still in RFCs proposed by the kernel community

fication time, providing a global scope as described in Section 5.1 of Chapter 5.

Chapter 5 also presents the design and implementation of our helper-rooted callgraph solution and proposes its integration with the verifier. In Chapter 6, we analyze our results by evaluating the generated callgraphs against runtime traces. At the end, Chapter 8 examines the approach's limitations in detecting dynamic nesting and explores potential improvements to the callgraph generation process.

1.3 Contributions

To summarize the main contributions, in this thesis we:

- Identify and demonstrate previously unknown safety issues caused by BPF program nesting, including stack overflows, deadlocks, and performance issues.
- Analyze existing ad-hoc solutions for preventing nesting problems and evaluate their limitations.
- Propose and implement a callgraph approach for static detection of unsafe nesting scenarios.
- Present a detailed design for integrating callgraph-based analyses into the BPF verifier to prevent unsafe program interactions.
- Provide an analysis of the static approach's limitations and outline future research directions.

1.4 Thesis Organization

Chapter 2 provides background on BPF programs and shows the importance of nesting through use cases and examples. Chapter 3 documents experiments showing how uncontrolled and unintended nesting of BPF programs can lead to stack overflows, deadlocks, and performance issues. Chapter 4 describes existing techniques to prevent nesting problems and discusses its limitations through examples. Chapter 5 presents the design and implementation of our callgraph solution, provides an overview of its integration with the verifier, and discusses the approach's limitations. Chapter 6 presents and evaluates results from our generated callgraphs and nesting detection component. Chapter 8 discusses important points related to our proposed solution and future work, while Chapter 9 presents our conclusion.

Chapter 2

Background

This chapter provides background on the need for kernel extensions in Operating Systems. It then discusses kernel extensions in the context of the Linux kernel, specifically focusing on BPF programs and their characteristics, and provides a high-level overview of the verifier, including its safety checks. The chapter then explains how safety plays a crucial role in BPF's widespread adoption. Finally, it introduces the concept of nesting in BPF programs and demonstrates why nesting is required through multiple use cases.

2.1 Background on Kernel Extensions

Kernel extensions provide a way to dynamically extend the kernel. This ability to change kernel functionality at runtime based on specific needs makes the system specialized. Users can load or unload these extensions based on their needs. However, buggy kernel extensions can compromise the kernel's stability and security. Safeguarding the kernel from faulty extensions has been a research focus since the '90s. Operating systems like SPIN [13] and VINO [34] explored different approaches - SPIN leveraging language safety through Modula-

3, while VINO employed Software Fault Isolation (SFI) based on transaction semantics.

Different commodity operating systems today provide different interfaces for extending their kernels. For example, Linux provides more expressive but less safe interfaces called kernel modules. Similarly, in macOS, kernel extensions were previously called kexts [3], which provided capabilities similar to kernel modules. After recognizing that unsafe kernel extensions could easily crash systems and were too risky for production environments, alternative approaches are being explored. For example, in Linux, developers turned to language safety for kernel modules by using type-safe languages like Rust [6] which can avoid some common mistakes but do not guarantee that the kernel won't crash, especially with "unsafe" semantics. MacOS took a different approach to safety by providing an interface to write kernel extensions in userspace, called SystemExtensions [3], trading expressiveness and performance for userspace safety.

Linux also introduced safer but less expressive kernel extensions called BPF programs, which rely on static analysis for their safety. In this work, we focus on BPF programs because of its wide adoption in the industry for various use cases, and the next section discusses how BPF programs work.

2.2 Berkeley Packet Filter

BPF programs are extensions to the Linux kernel. These programs are dynamically attached to specific hook points in the kernel during runtime, based on predefined program types. These hook points are triggered when encountered in the kernel execution path. Before attachment, BPF programs are verified by the in-kernel verifier, which performs static analysis to ensure safety and performance. The verification process involves two stages. In the first stage, the verifier performs DAG checks, such as ensuring no cycles (backward edges) exist

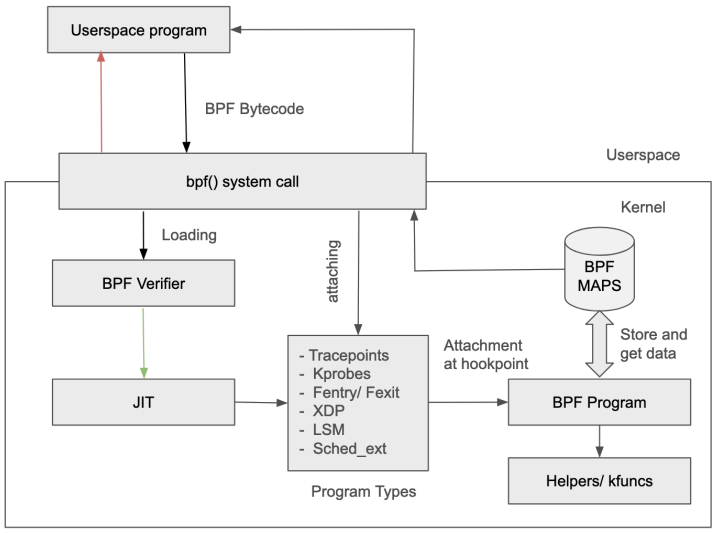


Figure 2.1: BPF Subsystem Overview

in the program. This verification step ensures that BPF programs have no infinite loops and are guaranteed to terminate. In the second stage, the verifier tracks all possible paths taken by a BPF program by performing register and stack tracking to ensure the program is not executing any invalid operations. During verification, the verifier examines each BPF program as an individual unit and makes assumptions about the runtime environment (kernel) with which it interacts. Once verified, these BPF programs are JIT compiled into machine instructions specific to the underlying architecture.

Figure 2.1 shows the architectural overview of the BPF subsystem. To load and attach a BPF program, the userspace program calls the BPF system call. Based on the provided program type and hook point, BPF programs are attached inside the kernel. BPF programs use maps to store data, which also enables communication with other BPF programs and userspace. BPF programs are triggered when they are part of the kernel execution flow.

Due to their strong safety guarantees, BPF programs have been adopted across a wide range of use cases. The initial adoption of BPF was in packet filtering for networking, which is

where the name Berkeley Packet Filters originated. Later, because of the safety guarantees and its dynamic nature led to BPFs[1] expansion into various use cases including tracing [5, 12], networking [10], security modules [11], scheduling [40], etc.

2.2.1 Helper Functions and kfuncs

To support various use cases, the BPF subsystem provides interfaces for kernel interaction: helper functions and kfuncs. Helper functions are stable interfaces that remain consistent between kernel releases, while kfuncs are not. Both helper functions and kfuncs must register and encode information such as arguments, return types, and compatible program types into the verifier to ensure BPF programs use these interfaces safely. The verifier assumes that execution within helper functions and kfuncs are inherently safe.

2.3 BPF Program Nesting

Nesting of BPF programs can occur when one BPF program triggers the execution of another BPF program. As shown in Figure 2.2, nesting could happen when a BPF program calls a helper function, which in turn invokes a kernel function to which another BPF program is attached. This type of interaction between multiple BPF programs is not only possible but often desirable for various use cases in the Linux kernel such as tracing and network stack debugging.

Tracing example: The dynamic and safe nature of BPF programs has made them ideal for tracing applications. Various tracing program types such as Tracepoints, Kprobes, and Trampoline (fentry/fexit) allow BPF programs to hook into specific points in the kernel, as

[1] As these programs are extended to adopt a wide range of use cases, they were renamed as extended Berkeley Packet Filters (eBPF). In this thesis, we use the terms BPF and eBPF interchangeably.

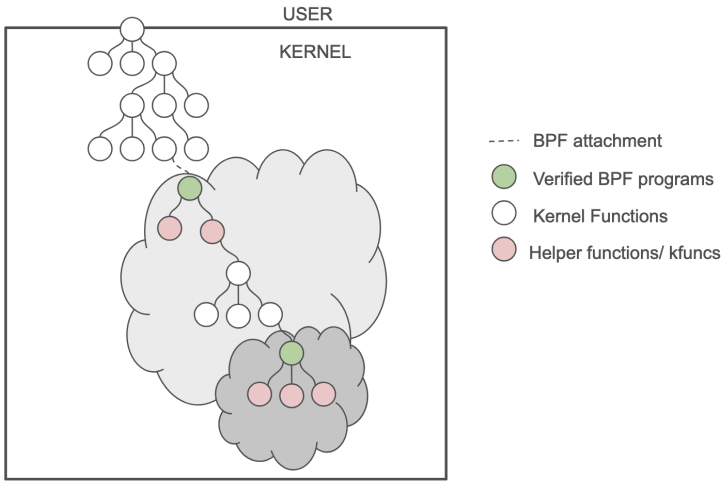


Figure 2.2: BPF Program Nesting

specified by these program types, to get tracing information. These hook points can also exist within helper function call graphs, allowing nested tracing scenarios.

For instance, as shown in Figure 2.3a, consider tracing the socket system call to log the Process ID (PID) and arguments passed into the system call. If the BPF program uses MAP-related helper functions to store, update, and lookup information for use by a userspace program, these map helpers internally employ spinlock functions to maintain synchronization between multiple map updates. Concurrently, if we use `trace_contention_begin, end` tracepoints to gather metrics about locks, allowing nesting enables us to trace both syscall events and lock events simultaneously.

Network stack debug example: packet, where are you? (pwru) is a networking debug tool [5] which provides insights about packet information such as socket buffer (skb), the CPU on which the process is running, the function containing the skb, and the process that triggered the event. To collect these metrics, pwru traces all functions that take skb as an input argument [20]. XDP/tc BPF program types, used for network packet filtering, firewalling, and load balancing, utilize helper functions and their callees, which often include

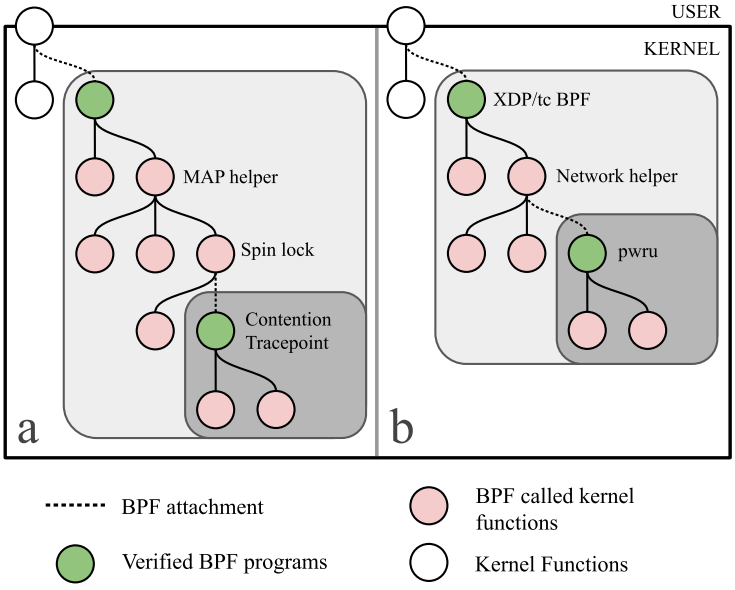


Figure 2.3: Nested BPF program usecases

skb as a function argument. In this scenario, tracing programs used by pwru are nested with XDP/tc programs, as shown in Figure 2.3b, to gather metrics, enabling users to debug issues.

2.4 Conclusion

To summarize, we have shown through two examples how BPF program nesting enables tracing and debugging, making it essential for modern use cases. However, uncontrolled nesting of BPF programs can lead to safety, performance, and security issues, which we discuss in the next chapter.

Chapter 3

Implications of uncontrolled nesting

This chapter presents how current verification methods fail to prevent stack overflows, deadlocks, and performance issues when BPF programs are nested. Through experimental results, we show that uncontrolled nesting of BPF programs can result in system crashes and performance issues, even though individual BPF programs are verified as safe. The experiments conducted demonstrate system crashes due to stack overflows, deadlocks, and performance problems caused by the overhead of nested BPF programs in the Linux kernel.

3.1 Stack overflow experiment

When BPF programs are attached to the kernel, they share and reuse the kernel stack. This design choice is based on the BPF verifier's assumption that by restricting each BPF program's stack usage to 512 bytes, and relying on helper functions to not consume a large amount of stack space, stack overflow should be prevented.

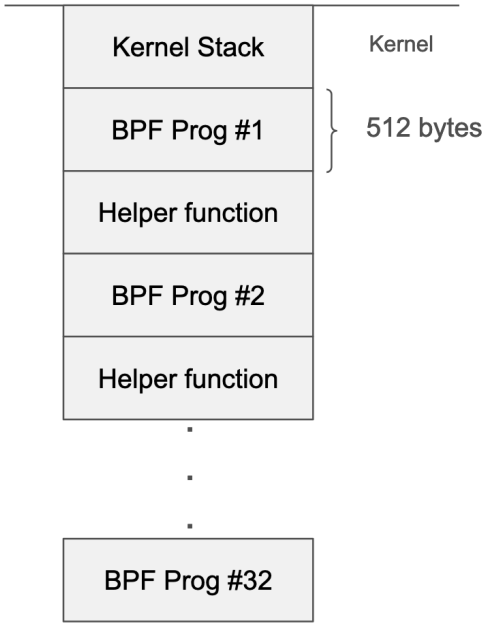


Figure 3.1: Stack overflow with nested BPF programs

The following two experiments demonstrate how uncontrolled nesting[1] can result in a kernel stack overflow. These findings were presented at LPC'23 [35].

Experiment 1: All BPF programs used in this experiment consume 512 bytes of stack space. As shown in Figure 3.2, we attached a BPF program to a kernel function. Within this program, we call a helper function. Subsequently, we attached 31 additional BPF programs to this helper function, with each program calling the same helper function[2]. When the kernel function to which our main BPF program is attached is triggered, it calls the helper function, which in turn sequentially triggers the 31 remaining BPF programs, as shown in Figure 3.1. This chain of nested calls results in a stack overflow, hitting the guard page and causing a kernel crash.

Experiment 2: This experiment demonstrates how just two BPF programs can overflow

[1] We expanded on a usecase of nested stacks in Section 8.1

[2] The necessity for 31 additional attachments stems from the recursion checks in fentry-type programs, which we discuss further in Section 4.1.2

```
SEC("fentry/{kernel_function}")
BPF_PROG_1(){
    // consume 512 bytes of stack space
    ...
    u32 pid = bpf_get_current_pid_tgid();
    ...
}

// attached 31 times
SEC("fentry/bpf_get_current_pid_tgid")
BPF_PROG_2(){
    // consume 512 bytes of stack space
    u32 pid = bpf_get_current_pid_tgid();
    ...
}
```

Figure 3.2: Sample BPF program to overflow the kernel stack

the kernel stack. Before getting into the details of the experiment, we provide an overview of tailcalls and BPF-to-BPF calls, which are important for understanding the context of this experiment.

BPF-to-BPF calls: These operate like regular function calls in BPF programs. When a BPF program invokes another function, it creates a new stack frame. The verifier ensures that the combined stack size of the BPF program and its callee functions does not exceed 512 bytes.

BPF Tailcalls: BPF Tailcalls function similarly to tailcall optimization in programming languages. The caller reuses the callee function's call stack, maintaining the 512-byte stack limit for a single BPF program. To prevent infinite recursion and stack overflow when used in conjunction with BPF-to-BPF calls (discussed below), the number of tailcalls is limited to 33.

In this experiment, we combined tailcalls and BPF-to-BPF calls to increase a BPF program's stack usage to 8 KB, as shown in Figure 3.3. When a BPF-to-BPF call creates a new stack

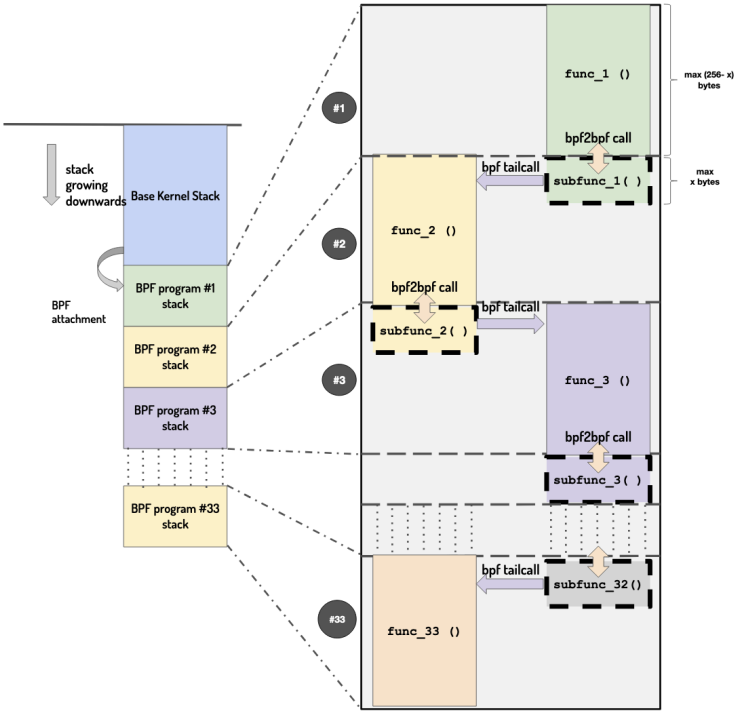


Figure 3.3: BPF program that consumes 8 KB of stack

frame on top of the BPF program, any tailcall within this function reuses the function's stack. Due to this interaction, the verifier restricts the BPF program stack size to 256 bytes when BPF-to-BPF calls and tailcalls are used together.

Using two such BPF programs, each with an 8 KB stack size, we showed that a single level of nesting is sufficient to overflow the kernel stack.

Takeaway: From both experiments, we showed that although BPF programs are individually verified to be safe, their nesting can still lead to kernel crashes.

3.2 Deadlock experiment

Deadlocks occur when multiple programs are unable to execute due to circular lock de-

```
1  // Attaching to a function inside the critical section of a map
2  SEC("fentry/_raw_spin_unlock_irqrestore")
3  BPF_PROG_2(){
4      __u32 value = 1;
5      bpf_map_push_elem(&map_queue, &value);
6  }
7
8  SEC("fentry/{kernel_function}")
9  BPF_PROG_1(){
10     __u32 value = 1;
11     bpf_map_push_elem(&map_queue, &value, 0);
12 }
```

Figure 3.4: Sample BPF program to overflow the kernel stack

pendencies or due to nested locking. The following two experiments demonstrate how the nesting of BPF programs can lead to two different deadlock scenarios. Later, we found that syzbot [7, 8, 9] raised similar issues in the mailing list.

Experiment 1: As shown in Figure 3.4, a BPF program calls a map helper function. Since map operations are atomic, they acquire and release a lock before and after any update, lookup, or delete operation. A second BPF program is attached to the critical section of the BPF map helper function. When this nested program calls the map helper function with the same map object, it causes a deadlock, stalling the CPU and crashing the kernel.

Experiment 2: We demonstrate an ABBA lock scenario caused by nested BPF programs. This setup involves four BPF programs. BPF programs 1 and 4 use map object A, while programs 2 and 3 use map object B. Similar to Experiment 1, programs 3 and 4 are attached inside the critical sections of map helpers, meaning program 3 is nested inside program 1, and program 4 is nested inside program 2. When a process on CPU 1 triggers BPF programs 1 and 3, while simultaneously a process on CPU 2 triggers BPF programs 2 and 4, an ABBA deadlock occurs. This happens because BPF programs 1 and 3 acquire locks on maps A and B respectively, while BPF programs 2 and 4 acquire locks on maps B and A respectively,

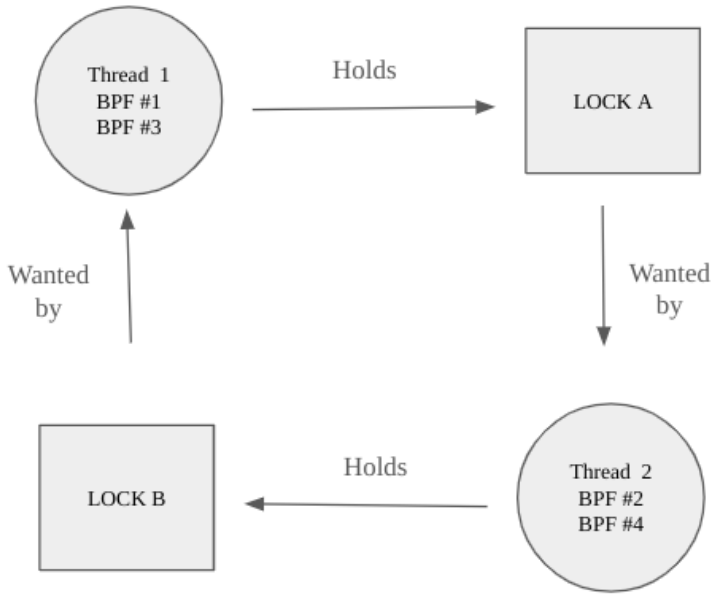


Figure 3.5: Deadlock Dependency Graph

creating a circular dependency, as shown in Figure 3.5.

Takeaway: From the both deadlock experiments we showed that despite BPF programs being individually verified as safe, their nesting can still cause kernel crashes.

3.3 Performance issues related experiment

Individually, BPF programs are expected to be short-lived [32]. However, nesting challenges this assumption, as even short-duration programs can introduce significant overhead that exceeds their individual execution times. This issue becomes particularly evident in complex production environments, where as many as 40 BPF programs often coexist, serving various purposes such as packet capturing, performance profiling, and security monitoring [18, 37], some of which are too slow to be on the data path.

Our experimental setup simulates such an environment, focusing on the interaction between

Redis	Mean Total Ops/s	Std Dev Total Ops/s	% Slowdown
Baseline	364628.14	12517.14	0.00
XDP	345417.99	5543.74	5.27
XDP Nested	298802.92	1700.49	18.05
Memcached			
Baseline	1006351.46	3144.78	0.00
XDP	998905.06	2311.21	0.74
XDP Nested	684187.57	2593.44	32.01

Table 3.1: Redis and Memcached throughput, measured as the total number of GET and SET operations performed per second

an XDP program and tracing BPF programs. The base scenario includes an XDP program that writes all source IP addresses to a map, incrementing the map value when a new IP matches an existing entry, simulating a common network monitoring use case. To simulate debugging the server, we attached multiple BPF programs to various kprobe and fentry hook points [18]. We tested that the programs have an overhead within acceptable limits [32]. The experiment introduces an unexpected nesting scenario where a BPF program is nested inside the XDP program. In this scenario, we implement a synthetic worst-case, long-running BPF program with a nested bpf_loop of 256 iterations, as shown in Figure 3.6. The results, presented in Table 3.1, show substantial performance impacts, with Memcached experiencing a 32.01% overhead and Redis an 18.05% overhead in throughput. This uncontrolled and unintended nesting of eBPF programs led to an undetected slowdown that could significantly impact production environments.

Takeaway: Even though BPF programs are guaranteed to complete quickly due to strict verifier restrictions, such as instruction limits, we demonstrated that nesting can inadvertently introduce expensive execution paths to the critical data path.

```
1  SEC("xdp")
2  BPF_PROG_1(){
3      // XDP program attached to network fast path
4      ...
5      __u64 *val = bpf_map_lookup_elem(&ip_map, &src_ip);
6      ...
7  }
8
9  SEC("fentry/__htab_map_lookup_elem")
10 BPF_PROG_2(){
11     // synthetic long running BPF program
12     bpf_loop();
13 }
```

Figure 3.6: Sample BPF program to show performance issues

3.4 Conclusion

From these experiments, we demonstrated that although BPF programs are individually verified as safe by the kernel, uncontrolled or unintended nesting of these programs can still lead to system crashes and significant performance issues.

Chapter 4

State of the Art

This section discusses existing checks in the Linux kernel that prevent common bugs arising from nested BPF programs. It discusses the problem of missed events, which is a byproduct of both these checks and preemption. It also covers the limitations of recently proposed solutions to prevent stack overflows, deadlocks, and performance problems caused by nesting.

4.1 General protections to prevent nesting problems

The BPF subsystem provides two types of checks to prevent uncontrolled nesting. The first type uses a per-CPU variable, which prevents recursion and re-entrancy. The second type uses a per-program variable, which prevents only recursion.

4.1.1 Problems with re-entrancy checks

In BPF, kprobes and tracepoints are program types used for tracing events. Both program types prevent recursion and re-entrancy by using the per-CPU variable `bpf_prog_active`.

This variable enforces a policy that allows only a single BPF program to run on a CPU at a time. This policy prevents stack overflow and deadlocks. However, it leads to missed events in two scenarios [38].

The first scenario involves nested programs. When two different BPF programs are nested, if the first program runs and calls the second, the second program won't be triggered due to the per-CPU policy, resulting in missed events. The second scenario is related to preemption. The increase in the number of missed events [23, 28, 29, 30] is a side effect of making BPF programs preemption-friendly for PREEMPT-RT kernels. For example, if a BPF program is triggered on a CPU and interrupted by another process with a different BPF program, the second program won't be triggered due to the per-CPU checks.

In summary, the use of tracing program types with a per-CPU variable can lead to missed events due to preemption, nesting, and recursion.

4.1.2 Problems with recursion checks

To address the challenges mentioned above and improve the performance of tracing events, BPF trampolines (fentry/fexit) were introduced. This BPF program type uses a per-program variable to prevent recursion. While this policy allows multiple BPF programs to nest, it does not include checks to prevent stack overflows and deadlocks, as discussed in Chapter 3. Along with these problems, it can miss events in recursive calls where a BPF program attached to a function is called inside a helper function. Preemption presents additional challenges with per-program checks. If a BPF program is preempted mid-execution and another thread scheduled on the same CPU triggers the same function containing the same BPF program, events can be missed.

The use of fentry program types with recursion checks can lead to stack overflows, deadlocks,

and missed events.

4.2 Missing of events is untenable

The missing of events, whether due to nesting, recursion, or preemption, can be catastrophic for security monitoring tools. Tools such as Hubble, Pixie, and Coroot [2, 4, 27] use tracing-type BPF programs to collect metrics and identify threats in the system. If events are lost due to nesting or preemption, it leads to inaccuracies in the collected metrics. In an attempt to prevent the missing of events due to preemption, the kernel community recently introduced new kfuncs: `bpf_preempt_{enable, disable}` and `bpf_local_irq_{save, restore}`. These functions, when used together, prevent BPF programs from being preempted or interrupted by another process or an interrupt. However, the problem of missed events due to nesting still persists.

4.3 Proposed runtime techniques are inadequate

For the problems described in Chapter 3 due to nesting, the kernel community has developed runtime techniques to prevent them. In this section, we describe the proposed runtime solutions and discuss their inadequacies.

Stack solution: As described in Section 3.1, the kernel stack can overflow with sufficiently deep nesting of BPF programs. To address this problem, the kernel community has recently proposed the concept of private stacks. This approach allocates BPF program stack space on the heap. However, helper function calls still consume space on the main kernel stack, as shown in Figure 4.1. As a result, uncontrolled and sufficiently deep nesting of BPF programs containing helper calls with large stack sizes can still overflow the kernel stack.

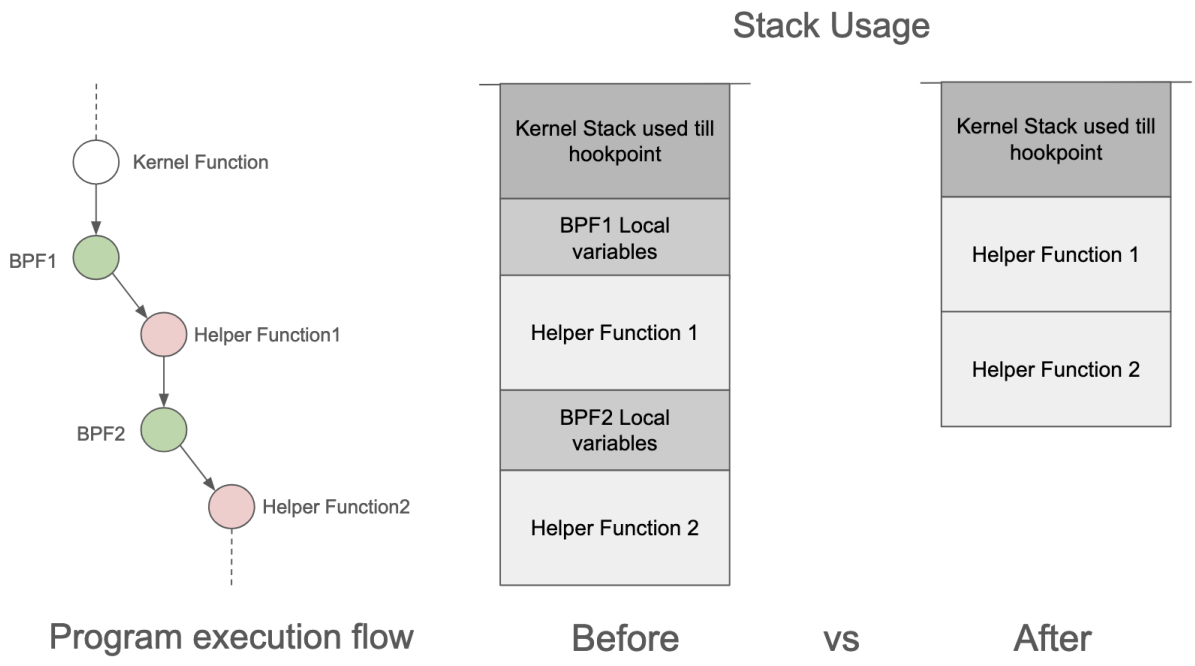


Figure 4.1: Kernel stack usage before and after the introduction of private stacks

Deadlock solution: Deadlocks are possible with BPF maps due to allowing re-entrancy, as demonstrated in Section 3.2. Two solutions have been proposed to prevent these issues, with one partially implemented in the kernel. The first solution avoids taking locks on the same map in nested scenarios, but this does not prevent ABBA deadlocks, as shown in Experiment 2 of Section 3.2. To address ABBA deadlock scenarios, the kernel community has recently proposed resilient spinlocks [39], which attempt to acquire locks for a specified time before giving up. Both solutions, however, result in missed map operations. As described in Section 4.2, missing such events could violate the consistency of monitoring tools that depend on maps for collecting events.

Performance solution: To identify performance issues, runtime metrics of BPF program execution can be obtained by enabling BPF_STATS_RUN_TIME. This macro can be enabled from userspace using the BPF syscall. The bpftop tool[1] builds on this functionality to

provide execution data for each BPF program. However, when multiple BPF programs are nested, there is no way to distinguish whether a program's overhead comes from its own execution or from the collective execution of nested programs.

4.4 Conclusion

In this chapter, we analyzed state-of-the-art solutions for preventing BPF nesting issues and identified their potential problems and side effects, showing that none provide adequate protection. The next chapter discusses why the current verifier implementation fails to prevent nesting problems and presents a system design that implements helper-rooted callgraphs and their future integration with the verifier to address these issues.

Chapter 5

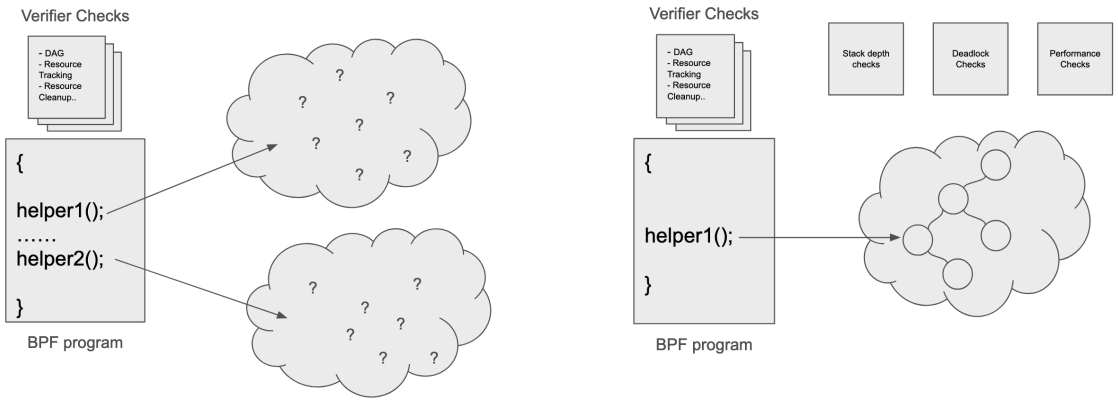
Design and Integration of Callgraphs

This chapter describes the root cause of the previously discussed nesting problems: the verifier's inability to understand how BPF programs interact as a whole. After establishing this motivation, we present helper-rooted callgraphs as a solution to provide the verifier with additional information about helper functions, enabling a global view. We then present our system architecture, describing in detail the components involved in callgraph generation and outlining how these components can integrate with the verifier to prevent nesting issues.

5.1 Design Motivation

5.1.1 Verifier's lack of global view

As discussed in Section 2.2, the verifier verifies each BPF program as an individual unit, making assumptions about its runtime interactions. As shown in Figure 5.1a, while the verifier can see the helper functions that BPF programs interact with, it cannot see the functions that these helper functions call. This limitation leads to not identifying that



(a) Current Verifier's View (b) Expanded Verifier's View

Figure 5.1: Verifier's Views: Current and Expanded

nesting happens through helper functions, as shown in Figure 2.2. This lack of a global view enables the nesting issues we discussed in Chapter 3.

5.1.2 Extending verifier view with helper-rooted callgraphs

We propose a helper-rooted callgraph approach to address the verifier's lack of a global view. As shown in Figure 5.1b, if the verifier has access to helper function callgraphs, it can update these callgraphs whenever a BPF program is attached. This allows the verifier to identify nested BPF programs and detect potential problems such as stack overflows, deadlocks, and performance issues.

5.2 Design Overview

As shown in Figure 5.2, our design uses helper-rooted callgraphs as its foundation and builds composite callgraphs as BPF programs are attached to the kernel. When a new

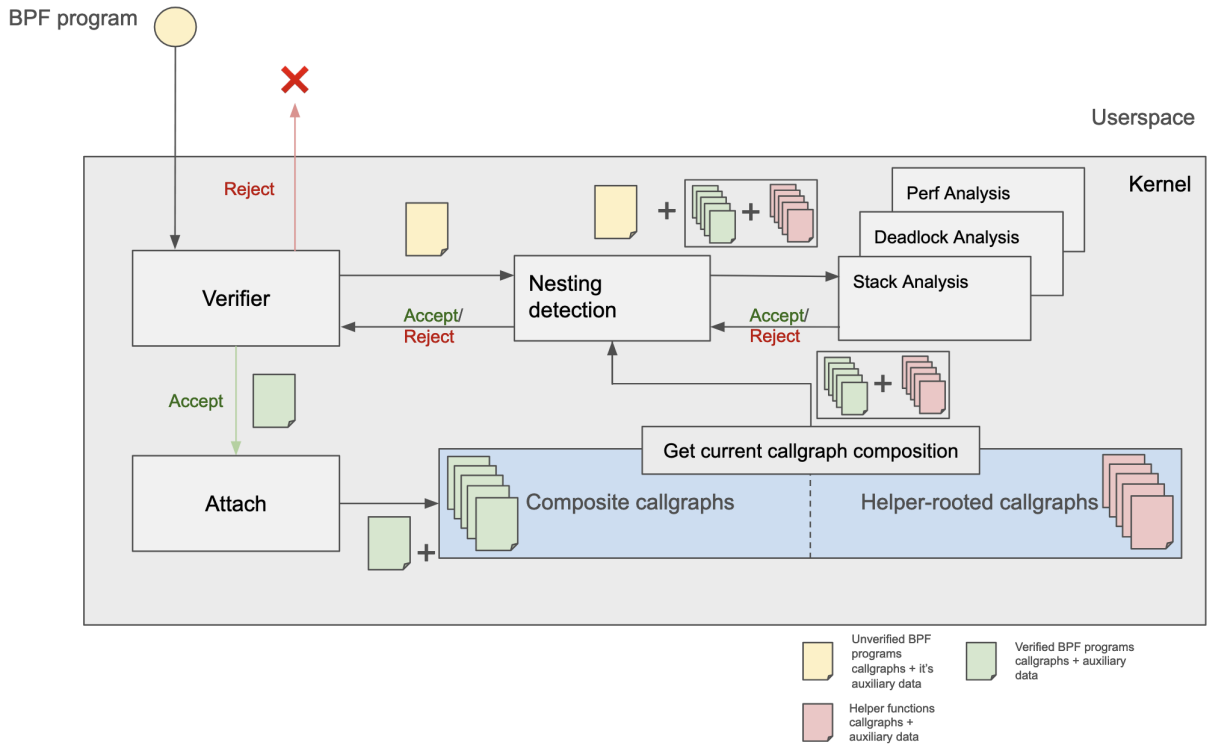


Figure 5.2: Design Overview

BPF program is loaded for verification, our system analyzes potential interactions with existing programs and detects nesting through these callgraphs. This analysis enables us to detect nesting scenarios and evaluate their safety implications by building analyses like stack overflow prevention, deadlock detection, and performance impact[1] on top of nesting detection.

5.2.1 Helper-rooted Callgraphs

Our goal is to statically generate callgraphs for helper functions in the Linux kernel to detect the nesting of BPF programs. This task is particularly challenging due to the kernel's extensive use of indirect calls for code modularity. Without resolving these indirect calls,

[1]More about these analyses is elaborated in Section 5.4

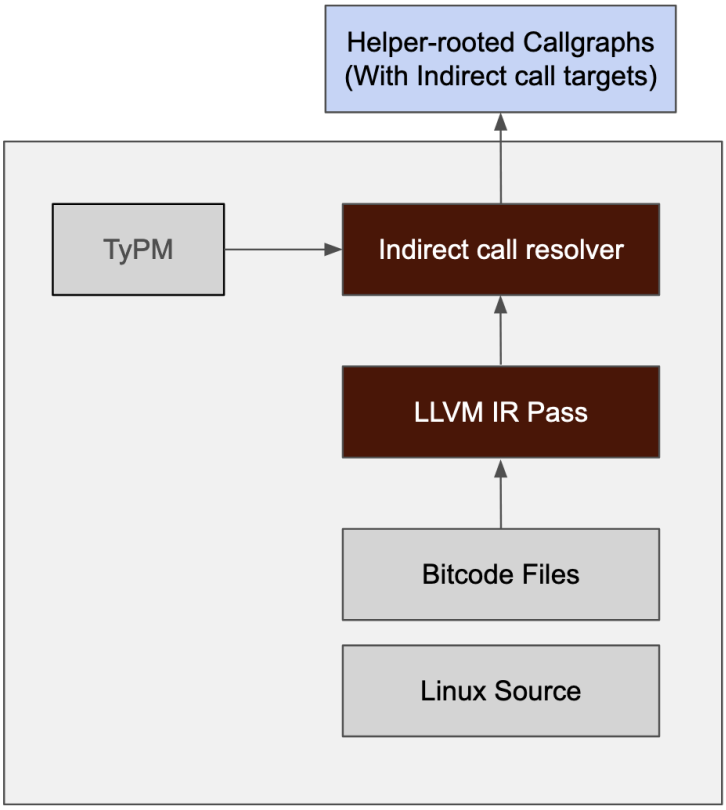


Figure 5.3: Design of Helper-rooted Callgraphs

the generated callgraphs would significantly underestimate potential execution paths.

To address this problem, we incorporate TyPM [26], a tool that identifies potential call targets through type matching across all kernel functions. These identified targets are then integrated into our callgraphs to provide comprehensive coverage of possible execution paths.

As shown in Figure 5.3, our design operates in two stages:

First Stage: In the first stage, after kernel compilation, the process begins by collecting the generated bitcode files. These files are then linked into a single bitcode file containing information about all kernel functions. An LLVM IR pass then generates callgraphs for all helper functions.

Second Stage: This stage focuses on resolving indirect calls. We execute TyPM on the

kernel to identify all indirect calls and their potential targets. These results are then parsed and integrated into the callgraphs generated in the first stage, providing a more complete view of possible execution paths.

5.2.2 Composite Callgraphs

The composite component maintains a current view of all attached BPF programs and their interactions on top of helper-rooted callgraphs. To generate the composite view, for each BPF program, we track three essential pieces of metadata:

1. Helper functions used by the program
2. Program attachment points in the kernel
3. Auxiliary data needed for safety analysis

When a new BPF program is attached, the dynamic component updates the current composition by recording the program's metadata, which includes its helper functions, kernel attachment point, and auxiliary data. The system then integrates this information into the existing callgraph composition by creating a new link between the attachment point and the BPF program. Through this updated composition, the system can identify nesting scenarios where the new program might interact with existing programs through shared helper functions or its callees/callers.

5.3 Callgraph integration with the verifier

In this section, we describe how the generated callgraphs can be integrated inside the verifier to detect and identify nesting problems.

As described in Figure 5.2, when a new BPF program is loaded, the verifier first performs its standard static analysis. Then, as a final verification step, it consults the Nesting Detection (ND) component to determine if the program's attachment is safe. The ND component gets the current composition of callgraphs and detects any BPF program nesting. If nesting is found, it runs analyses built on top of the ND component to detect potential nesting problems. If either the verifier's standard checks or the nesting analysis indicates potential problems, the program is rejected. Otherwise, the verifier accepts the program for attachment.

After successful verification and during program attachment, our callgraph system updates its current composition to incorporate the new program. This update includes adding the program's attachment point and helper function usage to the existing callgraph structure, along with auxiliary information required for ND program analysis. This updated composition then serves as the basis for ND to identify nesting problems when future programs are attached.

5.4 Building nesting checks on top of Nesting Detection

In this section, we explore how nesting checks can be built on top of the Nesting Detection component, as described in Section 5.3, to prevent stack overflows, deadlocks, and performance issues.

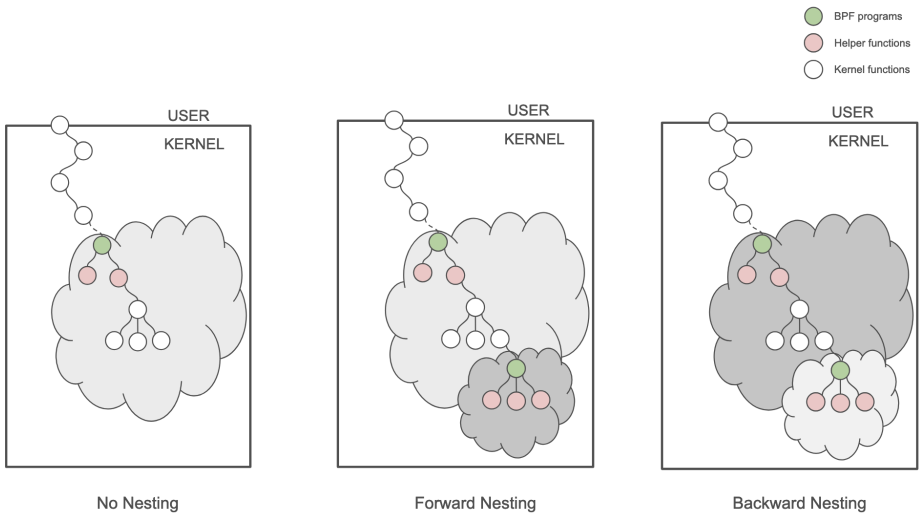


Figure 5.4: Nesting scenario's

5.4.1 Stack overflow checks

The Nesting Detection (ND) component will access helper-rooted callgraphs that update dynamically when new BPF programs are attached to the kernel. The helper-rooted and composite callgraphs must also contain information about the stack size consumed by each function.

In addition to Nesting Detection, the stack analysis will traverse the program's callgraph to identify nesting patterns and estimate total stack consumption. However, this approach requires modifications to how BPF programs interact with the kernel stack. Currently, BPF programs reuse the kernel stack, making it difficult to estimate the stack size consumed by a kernel thread before triggering a BPF program. Therefore, we propose implementing a clean stack state policy for BPF programs that defines strict limits on stack usage.

When analyzing callgraphs, three scenarios must be considered, as shown in Figure 5.4:

No nesting: A BPF program is attached to a kernel function without any nesting when a helper function is called.

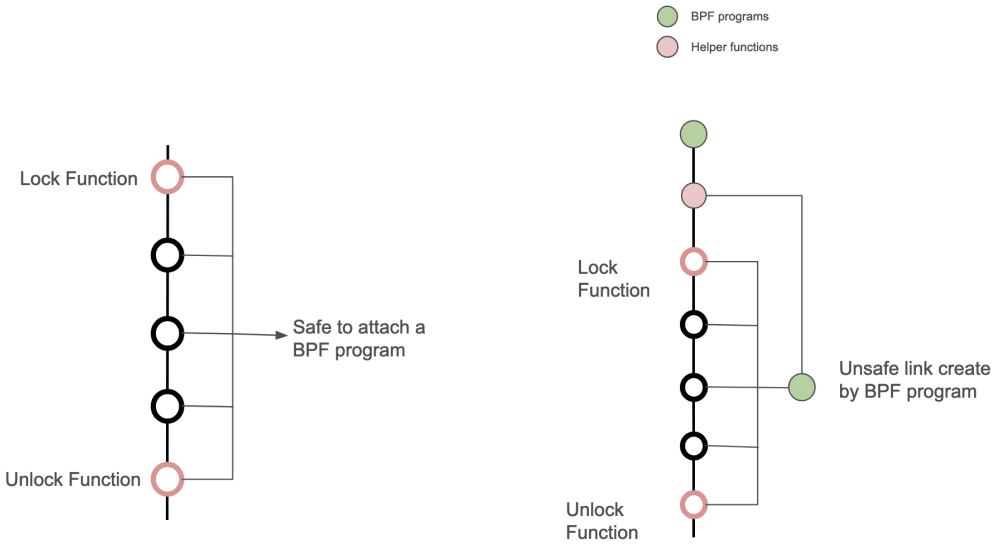


Figure 5.5: Deadlock scenario's

Forward nesting: A BPF program calls a helper function which, in turn, triggers another BPF program.

Backward nesting: A new BPF program is attached to a function called by an existing BPF program.

All three cases should be handled by the verifier's logic when walking and updating the new stack sizes. All three scenarios can be handled by performing a depth-first search on the graphs.

As a further optimization, in the case of backward nesting, one can traverse the callgraph in reverse to update the stack consumption of the existing BPF program. In all three scenarios, if the new total stack usage exceeds the limit, then the new BPF program is not safe to load.

5.4.2 Deadlock checks

Helper-rooted callgraphs can prevent deadlocks by identifying critical sections, similar to stack overflow checks. As discussed in Section 3.2, deadlocks in BPF programs occur when

a nested BPF program is attached to a function within a critical section and re-enters that critical section by acquiring a lock on the same kernel object, as shown in Figure 5.5. Since statically determining kernel object information is challenging, we propose using function-level information for static deadlock prevention.

A key question is how to identify functions within critical sections. One approach is to locate lock-taking functions through string matching of lock and unlock function names, then mark all functions between these points as being within critical sections. A more refined implementation would explicitly annotate all lock-taking functions in the kernel, which are relatively few in number.

5.4.3 Performance checks

Previous efforts to estimate BPF program runtime through static [32] and dynamic [1] approaches did not consider nesting scenarios in their solutions. As shown in Section 3.3, nesting of BPF programs can lead to serious performance problems. To address this, we propose adding helper-rooted callgraphs to existing static solutions to account for nesting. Like stack overflow checks, performance checks need to consider all three scenarios shown in Figure 5.4. The only difference between these two checks is that instead of stack sizes, we track runtime estimates of BPF programs.

5.5 Design Limitations

Our helper-rooted callgraph approach has two key limitations. First, the presence of loops in callgraphs complicates the collection of auxiliary data for analysis. Second, our approach to resolving indirect calls leads to overestimation of possible execution paths.

Handling Loops: Loops in callgraphs occur when a function directly or indirectly calls itself, creating a cycle in the call path. While loops do not significantly impact the analysis of control flow between functions, they present a critical challenge for collecting auxiliary data, for example, stack depth data required for stack analysis. When determining the stack depth of a function and its callees, the presence of loops can lead to incorrect calculations since recursive call paths could potentially continue indefinitely.

Indirect Call Resolution: Our approach to resolving indirect calls relies on TyPM, which uses function signature matching across all kernel functions. While this ensures we do not miss any potential call targets, it leads to an overestimation of possible execution paths. This conservative approach results in false positives where our system identifies potential nesting scenarios that may not occur in practice.

Loadable kfuncs: As discussed in Section 5.2.1, our helper-rooted callgraphs are generated from bitcode files produced during kernel compilation. However, this approach cannot handle kfuncs that are loaded into the kernel at runtime. Addressing this limitation might require a different solution. One potential approach would be to create a composite component similar to what we implemented for BPF programs. However, this would require information about all kernel functions since kfuncs can internally call any function in the kernel, including those not present in our current helper-rooted callgraphs.

Chapter 6

Results & Evaluation

This chapter presents the evaluation of the generated helper-rooted callgraphs. The first section of this chapter presents our experimental setup, followed by the results gathered from analyzing the generated callgraphs. Next, we describe our evaluation objectives, the methodology for testing the callgraphs in the context of nesting, and the results.

6.1 Evaluation setup

All experiments were run on a machine with an Intel Xeon Gold 5317 48-core CPU @ 3.00GHz and 256 GB of RAM. The system runs Linux kernel version 6.8.0 on Ubuntu 24.04.1 LTS. For generating the callgraphs, we used Linux kernel version 6.9.0 and LLVM version 15.

6.2 Analyzing the results from the callgraph

For the purpose of analyzing callgraphs, we collected callgraph information for all 278 helper functions in the kernel[1]. Although our callgraph tool was primarily designed to detect BPF program nesting, we extended it to generate data for all helper functions. For each helper function, we collected the data of all nodes in the callgraph, the number of indirect calls, the presence of loops, and the maximum call depth. The purpose of this analysis is to understand the generated callgraph information, identify discrepancies in the data, if any, and derive insights that could help us improve the callgraphs.

Of the 278 helper functions, our callgraph generator produced analyzable data for 162 helper functions, representing approximately 60% of the total. The remaining 40% of helper functions generated complex data due to the large number of nodes, indirect calls, and loops in their callgraphs. Analysis of randomly selected helper functions from this 40% revealed two main causes for the complexity. Common kernel functions that appear frequently in callpaths, such as RCU lock/unlock functions, printk-related functions, and panic-related functions, have deeper callgraphs and recurring occurrences in helper callgraphs. Furthermore, limitations in indirect call analysis created additional complexity. For instance, incomplete resolution by the TyPM tool led to unresolved indirect calls, while overestimation of predicted targets resulted in an excessive number of nodes. Due to these limitations in the produced data, our analysis focuses on the 60% of helper functions that produced reliable callgraph data.

Analysis: Our analysis of 162 helper functions showed that 105 helper functions have simple callgraphs with at most one node, indicating they execute directly without making additional function calls. These simple helper functions contain neither indirect calls nor loops. Among

[1]For this evaluation, kfuncs were not included

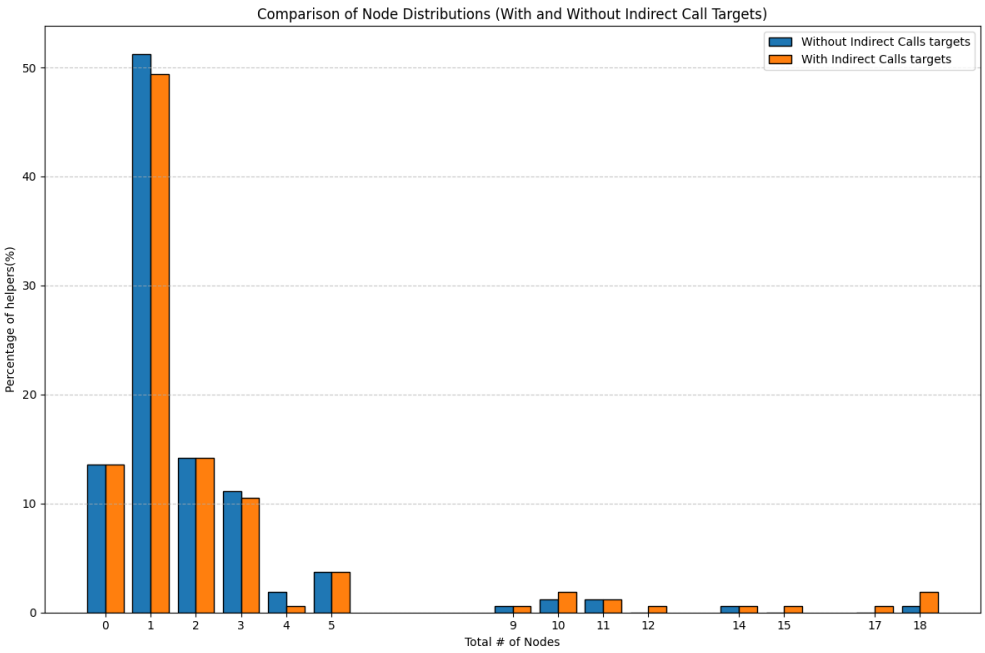


Figure 6.1: Graph node distribution across the helper functions

the remaining 57 helper functions, 54 contain at least one loop in their callgraphs, and 6 contain at least one indirect call. After including the predicted targets, the number of nodes in helper function graphs increased by at least 22%. The distribution in the change in the number of nodes before and after the inclusion of indirect calls is shown in Figure 6.1. This indicates that BPF programs using only this subset of 162 simple helper functions can benefit from our analysis, avoiding the complications of the more complex helpers we excluded.

6.3 Evaluation objectives

In our evaluation, we aim to answer the following questions:

1. **Soundness:** Does our helper-rooted callgraph approach catch every nested scenario?

2. **System impact:** What is the impact of the helper-rooted callgraph on system per-

Helper Function	# of Nodes in callgraph (Runtime Trace)	# of Nodes Not Found	Reason
bpf_task_storage_get_recur	3	0	
bpf_ringbuf_reserve	3	0	
bpf_ringbuf_submit	3	2	Static Calls
bpf_ringbuf_output	6	2	Static Calls
bpf_ringbuf_discard	3	2	Static Calls
bpf_sk_storage_get_tracing	12	0	
bpf_get_socket_ptr_cookie	1	0	
bpf_task_storage_delete_recur	2	0	
bpf_timer_init	4	0	
bpf_lru_pop_free	6	0	
bpf_timer_set_callback	1	0	
bpf_xdp_event_output	15	3	2 - Static Calls, 1 - Inlined Function
bpf_user_ringbuf_drain	7	2	Static Calls
bpf_bprintf_prepare	3	0	
bpf_timer_cancel	2	0	
bpf_map_lookup_elem	2	0	
bpf_map_update_elem	8	0	
bpf_timer_start	75	1	Inlined Function

Table 6.1: Analysis of helper functions and their trace information

formance and memory?

6.4 Evaluating helper-rooted callgraphs

The evaluation aims to verify whether our generated helper-rooted callgraphs can identify all possible nesting scenarios. We conducted an experiment that uses BPF program tests (Linux selftests). In this experiment, our goal is to collect runtime callee functions of helpers to evaluate the accuracy of our generated callgraphs.

To collect runtime traces of helper functions, we modified the kernel to enable tracing specifically for fentry/fexit BPF programs. We then gathered runtime call traces using ftrace while executing kernel self-tests for fentry/fexit. From these traces, we extracted the helper functions and their runtime callees. Through this process of running the self-tests, we were able to gather call traces for 18 helper functions.

The results from our experiments are presented in Table 6.1. Of the 18 helper functions, 12 showed clean matches between their runtime traces and our generated callgraphs.

Our analysis revealed limitations in handling static calls. Similar to indirect calls that are resolved at runtime, static calls use code patching to hard-code function pointers as jump instructions with zero overhead when not enabled. Neither our tool nor TyPM could resolve these static calls, resulting in seven unresolved static calls across five helper functions. Two additional discrepancies were found in `bpf_xdp_event_output` and `bpf_timer_start`. Although their callee functions (`bpf_xdp_copy_buf` and `avg_vruntime`) were present in the binary, they were not detected in our callgraphs due to function inlining in the helper calls.

Overall, our generated callgraphs were able to identify the functions generated by running the self-tests[2], with unresolved cases primarily due to unexpected static call handling, which remains as future work.

6.5 System impact for running callgraphs

While helper-rooted callgraphs help prevent nesting issues, it's important to understand their impact on system resources. To understand this, we evaluated the system impact of helper-rooted callgraphs in two key areas: memory consumption and runtime performance, if they are to be included in the verifier.

6.5.1 Memory Overhead

The helper-rooted callgraph approach requires storing additional information about function relationships. To understand this, we measured memory overhead in two ways:

Helper-rooted callgraphs memory usage: This experiment aims to show the memory

[2]These runtime traces also contain irq_enter_rcu and idle_cpu functions, which are not resolved in the callgraphs. These functions are ignored in our analysis because they are part of interrupts.

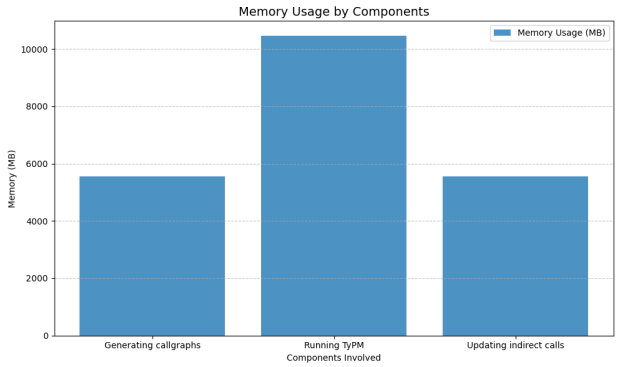


Figure 6.2: Memory consumption while running each component in the callgraph

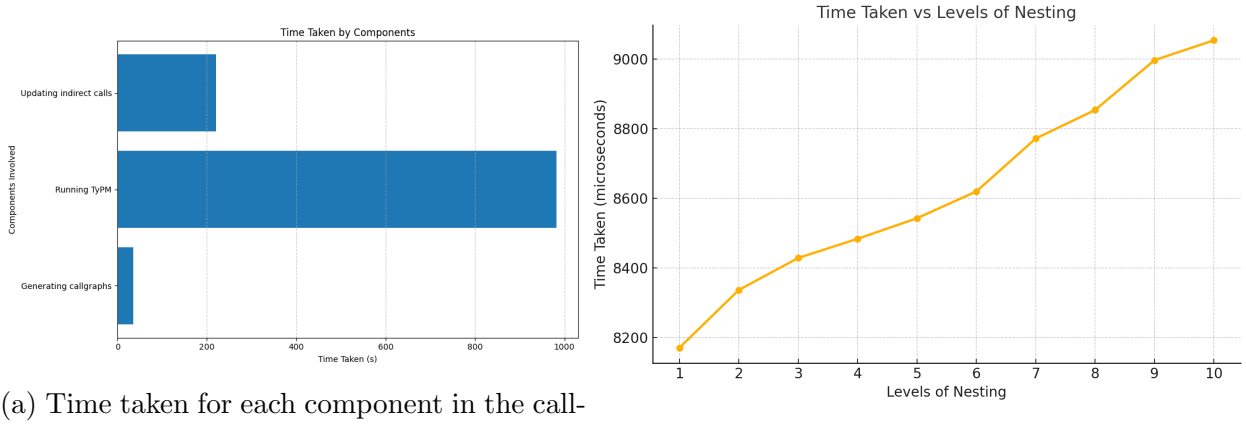
overhead during the generation of the callgraphs. The results from the experiments are shown in Figure 6.2. This includes all three stages: initial callgraph generation, processing of indirect call targets using TyPM, and integration of these targets into the initial callgraphs. As shown in Figure 6.2, memory usage doubles during TyPM processing because TyPM needs to analyze the entire kernel to predict indirect call targets.

Composite graphs memory usage: This experiment aims to show the memory overhead when updating the graphs with BPF programs and detecting the nesting. Through our experiments, we found that the memory usage has been nearly constant with different levels of nesting.

6.5.2 Performance Overhead

We measured the performance impact of both the helper-rooted callgraphs and composite callgraphs components in our callgraph solution:

Helper-rooted Callgraphs: As shown in Figure 6.3a, we measured the time consumed by the static component of our callgraph generation process. Similar to the memory usage experiment, we divided this process into three stages. Our experiments show that TyPM pro-



(a) Time taken for each component in the callgraph

(b) Time taken for detecting nesting

Figure 6.3: Comparison of performance metrics

cessing accounts for 79% of the total execution time, representing the most time-consuming phase.

Composite graphs: This experiment measures the performance overhead of our dynamic component, adding BPF programs to callgraphs and generating nesting information. As shown in Figure 6.3b, we evaluated the time taken to process BPF programs at different nesting levels. This measurement demonstrates how the performance overhead scales with increasing nesting depth. The results show that processing time increases linearly with nesting depth, which is expected, as each additional level of nesting requires analyzing one more set of program interactions.

Chapter 7

Related Work

Kernel extensions allow the operating system to be customized and extended, but they introduce challenges related to performance and safety. In this section, we review prior work on improving kernel extension performance, stack management in kernel extensions, and callgraph analysis techniques relevant to the nesting problem.

7.1 Improving Kernel Extension Performance

Performance is a critical concern in kernel extensions, as they operate within the kernel's execution path and can significantly impact overall system performance. Early operating systems like SPIN [13], VINO [33], and Exokernel [19] explored methods to safely extend kernel functionality while minimizing performance overhead. SPIN used language-based protection mechanisms to allow safe customization without sacrificing performance. It provided fine-grained interfaces to kernel services. VINO implemented grafting and software fault isolation for safe kernel extensions. It used lightweight transactions to track resources and abort misbehaving extensions. Exokernel allowed applications to implement operating

system abstractions at the application level, reducing kernel overhead. It provided secure bindings to allow applications direct access to hardware resources. However, none of these systems specifically addressed performance impacts due to the nesting of extensions.

Several recent works have identified performance and security issues arising from the execution of BPF programs. Sahu et al. [32] proposed a runtime estimation mechanism for BPF programs but did not specifically address nesting-related performance degradation.

7.2 Stack Management in Kernel Extensions

Efficient stack management is crucial in kernel extensions due to limited stack sizes and the risk of stack overflows, which can lead to system crashes or security vulnerabilities. SPIN [13] focused on extensibility while maintaining safety through language-based mechanisms. Chiueh [16] proposed integrating segmentation and paging protection for safe kernel extensions. Wahbe et al. [41] presented software-based fault isolation (SFI) to sandbox untrusted code, preventing it from affecting other parts of the system. While these works enhance safety and performance in kernel extensions, they do not specifically address stack management challenges associated with nested functions.

To enhance robustness in kernel extensions, several studies have explored various strategies. Seltzer et al. [34] discussed strategies for surviving misbehaved kernel extensions. However, they did not provide mechanisms to prevent stack overflows caused by nested functions. Lu et al. [24] proposed MOAT, a system that uses Intel Memory Protection Keys (MPK) to isolate BPF programs from the kernel and each other. While MOAT enhances overall BPF security through hardware-enforced memory isolation, it does not specifically address stack management challenges or detection of potential overflows from nested function calls in BPF programs. Jia et al. [22] argued that kernel extension verification is untenable for complex

BPF programs, highlighting challenges in ensuring safety properties, including stack safety, through static analysis. However, they did not offer practical solutions to this problem.

To estimate stack usage accurately, the LLVM community developed a static stack depth analysis tool [15] for the Zircon kernel. This tool computes the worst-case stack depth by extracting callgraphs and stack frame sizes, eliminating guesswork in memory allocation. However, it is tailored for Zircon and does not address the specific challenges of BPF program nesting within the Linux kernel.

Our work addresses these gaps by detecting potential stack overflows caused by nested execution in BPF programs. By constructing callgraphs and analyzing stack usage along potential callpaths, we identify nesting scenarios that may lead to stack exhaustion. This proactive detection allows developers to mitigate stack management issues before deploying their BPF programs.

7.3 Callgraph Analysis

Callgraph analysis is a fundamental technique for understanding program structure by representing the calling relationships between functions. It has been extensively used in compiler optimizations, program comprehension, and various static analysis tools. Early work by Ryder [31] introduced static callgraph construction for procedural languages, enabling optimizations such as inlining and dead code elimination. Callahan et al. [14] extended this by constructing procedure call multigraphs, capturing multiple calls between procedures to provide a more detailed view of program interactions. Grove et al. [21] developed algorithms like Class Hierarchy Analysis and Rapid Type Analysis to improve callgraph precision in the presence of dynamic dispatch and inheritance.

In kernel code analysis, callgraphs have been employed for various purposes. Cho et al. [17] used function callgraphs to evaluate code coverage for kernel fuzzers, enhancing fuzzing effectiveness by identifying critical execution paths. However, since fuzzing cannot explore all execution paths and relies on dynamic analysis, their approach is unsuitable for statically detecting nesting scenarios in BPF programs.

Lu [25] proposed Type-Based Dependence Analysis (TyPM) for program modularization, leveraging callgraphs to understand dependencies in large codebases like the Linux kernel. Our approach differs by utilizing callgraph analysis combined with indirect call predictions to improve the accuracy of our static analysis of BPF programs within the kernel.

Chapter 8

Discussion & Future Work

In this chapter, we discuss the cases where the callgraph solution cannot detect the nesting of BPF programs and conclude with how helper-rooted callgraphs can be improved by providing context-specific information.

8.1 Dynamic nesting by replacing indirect calls

Our previous discussion focused on BPF programs attached to kernel functions. The introduction of BPF_STRUCT_OPS enables a new form of interaction: replacing indirect calls with BPF programs. This program type has broad applications, from implementing congestion control to scheduling algorithms. We define "dynamic nesting" as scenarios where a BPF program calls these program types.

A good example of dynamic nesting appears in Sched-ext, a BPF-based scheduling framework that enables safe and dynamic adaptation of scheduling policies based on system workloads. To enforce these policies, BPF programs can be attached to cgroups. In scenarios involving

a hierarchy of cgroups (parent cgroups with multiple and nested child cgroups), scheduling-based BPF programs can be attached to all cgroups present in the hierarchy. To obtain scheduling information from child cgroups, parent BPF programs need to invoke the child BPF programs. This process allows for the collection of scheduling information across the cgroup hierarchy [36, 40]. With a call graph, we cannot precisely determine which BPF program could be triggered, leading to over-approximation, but we need more information about the object that triggers that BPF program. To avoid uncontrolled nesting, safety measures should be taken to avoid calling the same struct-ops with the same object, which can lead to recursion.

8.2 Improving call graph for helper functions

In Section 6, we showed that some helper functions have deep call depths and numerous indirect calls in their generated callgraph data. Additionally, TyPM, the tool that we used for predicting indirect call targets, tends to overestimate potential targets, resulting in overly large callgraphs. To improve our current approach, we could incorporate context-specific information to filter TyPM's predicted targets. For example, BPF programs use different types of maps for different scenarios. When a map-related helper function is called, it invokes functions from `bpf_map_ops` based on the passed object. Since BPF supports only a limited set of map types, hardcoding map operation types with their specific potential targets could significantly reduce overestimation and improve callgraph accuracy.

This map-specific optimization shows a broader opportunity: incorporating context-related information could help refine callgraphs. By applying similar context-aware filtering in other areas, we could further improve the precision of our callgraph analysis.

Chapter 9

Conclusion

In this thesis, we explored the growing relevance of BPF programs and their increasing adoption in industry due to their strong safety properties. We demonstrated that the nesting of BPF programs, which is necessary for certain tracing and networking use cases, can lead to serious problems. Through experiments, we showed how uncontrolled nesting can cause stack overflows, deadlocks, and performance issues.

After exploring state-of-the-art techniques for preventing these issues and their limitations, we proposed a static analysis approach using callgraphs to detect nesting. We presented a design for integrating these callgraphs into the kernel's verification process to prevent unsafe nesting scenarios. Our evaluation revealed that while this solution works effectively for helper functions with clear callgraphs and resolved indirect calls, it is limited to a subset of all helper functions.

Finally, we discussed the limitations of our approach and proposed potential improvements for future work, including better handling of indirect calls and implementing different types of analysis on top of the Nesting Detection component.

Bibliography

[1] bpftop. https://github.com/Netflix/bpftop.

[2] coroot. https://github.com/coroot/coroot.

[3] Implementing drivers, system extensions, and kexts. https://developer.apple.com/documentation/kernel/implementing_drivers_system_extensions_and_kexts/.

[4] pixie. https://github.com/angelolab/pixie.

[5] pwru (packet, where are you?). https://github.com/cilium/pwru.

[6] rust-for-linux. https://github.com/rust-for-linux.

[7] possible deadlock in htab_lru_map_delete_elem. https://lore.kernel.org/bpf/CAADnVQJVJADKw0KC6GzhSOjA8DJFammARKwVh+TeNAD7U3h91A@mail.gmail.com/T/, .

[8] possible deadlock in __queue_map_get. https://lore.kernel.org/lkml/0000000000004c3fc90615f37756@google.com/, .

[9] possible deadlock in __bpf_ringbuf_reserve. https://lore.kernel.org/lkml/0000000000004aa700061379547e@google.com/, .

[10] Cilium, May 2024. URL https://cilium.io.

[11] Lsm bpf programs. https://docs.kernel.org/bpf/prog_lsm.html, May 2024.

[12] Tetragon. https://github.com/cilium/tetragon, May 2024.

[13] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the spin operating system. *SIGOPS Oper. Syst. Rev.*, 29(5):267–283, December 1995. ISSN 0163-5980. doi: 10.1145/224057.224077. URL https://doi.org/10.1145/224057.224077.

[14] David Callahan, Alan Carle, Mary Wolcott Hall, and Ken Kennedy. Constructing the procedure call multigraph. *IEEE Trans. Softw. Eng.*, 16(4):483–487, April 1990. ISSN 0098-5589. doi: 10.1109/32.54302. URL https://doi.org/10.1109/32.54302.

[15] Annie Cherkaev. static stack depth analysis tool. https://lists.llvm.org/pipermail/llvm-dev/2018-July/124572.html, 2018.

[16] Tzi-cker Chiueh, Ganesh Venkitachalam, and Prashant Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. *SIGOPS Oper. Syst. Rev.*, 33(5):140–153, December 1999. ISSN 0163-5980. doi: 10.1145/319344.319161. URL https://doi.org/10.1145/319344.319161.

[17] Mingi Cho, Hoyong Jin, Dohyeon An, and Taekyoung Kwon. Evaluating code coverage for kernel fuzzers via function call graph. *IEEE Access*, 9:157267–157277, 2021. doi: 10.1109/ACCESS.2021.3129062.

[18] Jonathan Corbet. Bpf at facebook (and beyond), 2019. URL https://lwn.net/Articles/808048/.

[19] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, page 251–266, New

York, NY, USA, 1995. Association for Computing Machinery. ISBN 0897917154. doi: 10.1145/224056.224076. URL https://doi.org/10.1145/224056.224076.

[20] Martynas Pumputis Gray Liang. pwru - Linux kernel and BPF-based networking debugger. https://lpc.events/event/18/contributions/1942/, 2024.

[21] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '97, page 108–124, New York, NY, USA, 1997. Association for Computing Machinery. ISBN 0897919084. doi: 10.1145/263698.264352. URL https://doi.org/10.1145/263698.264352.

[22] 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*, HOTOS '23, page 150–157, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701955. doi: 10.1145/3593856.3595892. URL https://doi.org/10.1145/3593856.3595892.

[23] Jianlin-lv. KProbes Missing Issue. https://github.com/iovisor/bcc/issues/4198, August 2022.

[24] Hongyi Lu, Shuai Wang, Yechang Wu, Wanning He, and Fengwei Zhang. MOAT: Towards safe BPF kernel extension. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 1153–1170, Philadelphia, PA, August 2024. USENIX Association. ISBN 978-1-939133-44-1. URL https://www.usenix.org/conference/usenixsecurity24/presentation/lu-hongyi.

[25] Kangjie Lu. Practical program modularization with type-based dependence analysis.

In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1256–1270, 2023. doi: 10.1109/SP46215.2023.10179412.

[26] Kangjie Lu. Practical program modularization with type-based dependence analysis. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1256–1270, 2023. doi: 10.1109/SP46215.2023.10179412.

[27] Yu Luo, Kirk Rodrigues, Cuiqin Li, Feng Zhang, Lijin Jiang, Bing Xia, David Lion, and Ding Yuan. Hubble: Performance debugging with In-Production, Just-In-Time method tracing on android. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 787–803, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/luo.

[28] John Mazzie. Tracing NVMe Driver with BPF missing events. https://lore.kernel.org/all/CAPxVHdL-dT2GQh-HEkNjNoTEzA9DRL4W4ZfmUzc1+Bdz89fftQ@mail.gmail.com/, May 2022.

[29] netedwardwu. memleak tool misjudge memory leak because of the hardware interrupt. memleaktoolmisjudgememoryleakbecauseofthehardwareinterrupt, 2020.

[30] Jiri olsa. where have all the kprobes gone. https://lpc.events/event/17/contributions/1609/, November 2023.

[31] Barbara G. Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, SE-5:216–226, 1979.

[32] Raj Sahu and Dan Williams. Enabling bpf runtime policies for better bpf management. In *Proceedings of the 1st Workshop on EBPF and Kernel Extensions*, eBPF '23, page 49–55, New York, NY, USA, 2023. Association for Computing Machinery.

ISBN 9798400702938. doi: 10.1145/3609021.3609297. URL https://doi.org/10.1145/3609021.3609297.

[33] Margo I Seltzer, Yasuhiro Endo, Christopher A Small, and Keith Smith. Vino: The 1994 fall harvest. 1994.

[34] 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.

[35] Siddharth Chintamaneni, Sai Roop Somaraju, Dan Williams. Overflowing the kernel stack with BPF. https://lpc.events/event/17/contributions/1595/, November 2023.

[36] Yonghong Song. bpf: Support private stack for bpf progs. https://lore.kernel.org/bpf/ec48e1b2-ff1d-499b-8ada-ba76a4bae9bb@linux.dev/T/, 2024.

[37] Alexei Starovoitov. Bpf at facebook, 2019. URL https://kernel-recipes.org/en/2019/talks/bpf-at-facebook/.

[38] Alexei Starovoitov. bpf: Use prog-¿active instead of bpf_prog_active for kprobe_multi. https://lore.kernel.org/bpf/YqCW+EgPjQ+fS0WW@krava/T/#mc07686fda9b7b6ff7df6bbc2d951da23a09fcc74, 2022.

[39] Alexei Starovoitov. More features and use-cases for sched_ext. In *LSM/MM/BPF Summit 2024*, May 2024.

[40] David Vernet. More features and use-cases for sched_ext. In *LSM/MM/BPF Summit 2024*, May 2024.

[41] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. *SIGOPS Oper. Syst. Rev.*, 27(5):203–216, December

1993. ISSN 0163-5980. doi: 10.1145/173668.168635. URL https://doi.org/10.1145/173668.168635.