On the implications of unsafe eBPF composition

Overflowing the kernel stack with eBPF

Sai Roop Somaraju

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 Engineering

Daniel Williams, Co-chair
Creed Jones, Co-chair
Chang Woo Min

May 6, 2024
Blacksburg, Virginia

Keywords: Extended Berkeley Packet Filter, Linux, Stack Overflow

On the implications of unsafe eBPF composition

Overflowing the kernel stack with eBPF

Sai Roop Somaraju

ABSTRACT

In the era of Linux being omnipresent, the demand for dynamically extending kernel capabilities without requiring changes to kernel source code or loading kernel modules at runtime is increasing. This is driven by numerous use cases such as observability, security, and networking, which can be efficiently addressed at the system level, underscoring the importance of such extensions. Any extension requires programmers to possess high levels of skill and thorough testing to ensure complete safety. The eBPF subsystem in the Linux kernel addresses this challenge by allowing applications to enhance the kernel's capabilities at runtime, while ensuring stability and security. This guaranteed safety is facilitated by the verifier engine, which statically verifies BPF code. In this thesis, we identify that the verifier implicitly relies on safety assumptions about its runtime execution environment, which are not being upheld in certain scenarios. One such critical aspect of the execution environment is the availability of stack space for use while executing the BPF program. Specifically, we highlight this fundamental issue in certain configuration of the BPF runtime environment within the Linux kernel and how this unsafe composition allowed for kernel stack overflow, thus violating safety guarantees. To tackle this problem, we propose a stack switching approach to ensure stack safety and evaluate its effectiveness.

On the implications of unsafe eBPF composition

Overflowing the kernel stack with eBPF

Sai Roop Somaraju

GENERAL AUDIENCE ABSTRACT

Many platforms worldwide, including Meta, Netflix, Google, Cloudflare, and others, rely on the Linux kernel to manage their servers. To ensure system security, improve monitoring, and enhance networking efficiency, various kernel capabilities are dynamically added or removed at runtime without the need for reboots, thus minimizing downtime for users. The Linux Extended Berkeley Packet Filter (eBPF) subsystem facilitates dynamic and safe extension by securely verifying the code injected into the kernel. This eases server maintenance tasks, eliminating concerns about system crashes when making runtime changes as eBPF is guaranteeing safety at all times. In our research, we demonstrate that if we attach verified eBPF in a certain manner, we can potentially stack overflow the kernel stack and crash the whole kernel due to unsafe composition with the Kernel. We also propose two solutions to this problem, which can ensure that eBPF remains safe while adhering to the guarantees it provides.

Acknowledgments

To my esteemed committee members, I extend my deepest gratitude for your unwavering support and insightful guidance. Especially to my advisor, Dr. Williams, your dedication to fostering a research mindset in students truly inspired me.

I'm deeply grateful to my dear lab mates, Sidharth and Raj, and my fantastic roommates, including Bhanu, who were there for me with unwavering support every step of the way on my MS journey.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Traditionally, the operating system kernel has been the ideal place to add fundamental features like security, monitoring, and networking due to its privileged control over system resources. However, the kernel can be slow to evolve in adopting new features compared to advancements happening outside the kernel. This is because its central role demands exceptional stability and security, which can be compromised by hasty additions. The advent of the eBPF subsystem changes this dynamic [5]. eBPF guarantees the safety and security of the kernel for any extensions loaded from user space [10]. This assured safety stems from the eBPF verification engine Fig.1.1, which meticulously examines user-space programs written in C. The verifier establishes critical properties for these programs before they enter the kernel runtime, including guaranteed termination, memory safety, and information flow security [8].

In the eBPF subsystem after the verifier makes all checks and deems a program is safe, it is expected that these safety properties are upheld or maintained during the lifetime of the eBPF runtime execution [6]. For example, One such runtime property is the availability of the clean and safe stack space for eBPF to execute its functions. Currently, the eBPF subsystem guarantees and checks that a program will not exceed a max stack depth. However, We will

show that in certain cases these guarantees are not extended to its composition with kernel.

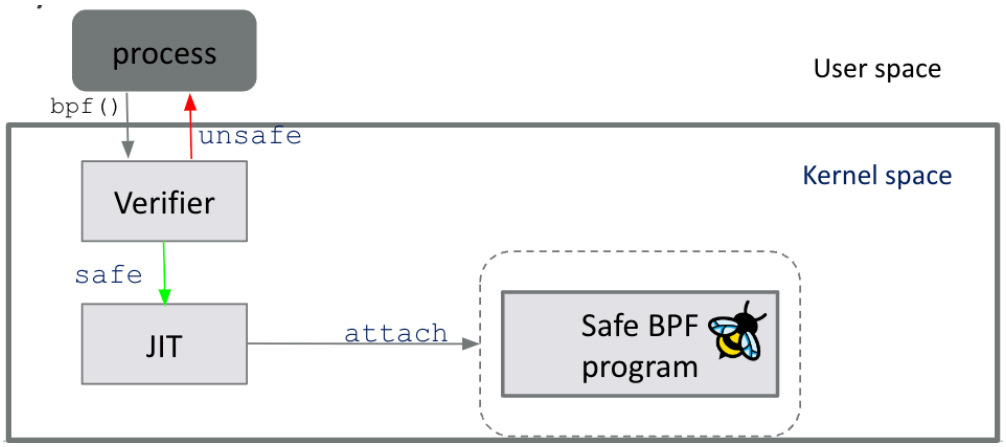


Figure 1.1: Verifier engine which statically verifies eBPF code guaranteeing safety.

In our research we analyse the current implementation of the eBPF subsystem where we found two unknowns that BPF is unaware of while it executes the attached BPF program *the composite stack start state* of BPF execution, and *the composite stack requirement* of BPF execution Fig.4.4.

1.1.1 Composite stack start state

Firstly, upon every event triggering the attached BPF program, it inherits the same kernel stack as any thread running in kernel mode. Kernel stacks are limited, and any overflows can be fatal for the system. The question arises about the situation when the kernel stack is already loaded and leaves less space than the stack space required for the BPF execution.

1.1.2 Composite stack requirement

Secondly, BPF, in order to expand its very limited functionality, utilizes components like helper functions, BPF tail calls, etc., to interact with other kernel subsystems in order to improve expressiveness. One interesting aspect is that helper functions called from BPF

programs serve as gateways for BPF programs and are not verified by the eBPF verifier[1]. It can also be inferred that anything happening inside the helper functions can affect BPF safety, of which BPF remains unaware. In the context of stack space requirements, it is impossible to estimate the correct BPF composite stack size without an analysis of helper functions.

In this thesis, we develop methodologies to demonstrate the insufficiency of the verifier checks by leveraging these two issues for kernel stack overflow using verified safe eBPF programs. Additionally, to address these challenges while ensuring BPF safety, we propose a solution where we switch stacks on speculating stack overflow conditions and we also evaluate stack switch solution with various performance benchmarks.

1.2 Contributions

In this thesis we identify two fundamental problems in the setup of the BPF runtime environment that allowed us to overflow the kernel stack which violates the safety guarantee that eBPF promised. In such scenarios to provide stack guarantees, we present a solution which uses a clean stack for the eBPF execution.

Summary of the contributions:

- Identified eBPF stack memory composition fails with kernel stack memory by using different dynamic and static methodologies.
- Proposed a new stack model for eBPF programs and based upon the design system requirements. In our solution we do stack switching, where kernel has a pre-allocated stack space per CPU for optimal memory usage.

[1]Unlike verifying the limited BPF program instructions, verifying helpers would entail verifying the entire Linux kernel, which is not practical at this point.

- We also propose some more stack overflow cases which happen inside the helper functions due to recursion as part of future work.

1.3 Thesis Organization

Chapter 2 introduces concepts of Extended Berkeley Packet Filtering (eBPF) implemented in the Linux Kernel and static verification. Here we also talk about the eBPF verifier, helper functions, bpf2bpf calls and bpf tail calls. Chapter 3 describes the relevant related work done on composite runtime issues that can arise when the execution requirement is unknown during start of the service. Since this is a new and evolving area lot of the state of the art work is being done in industry so we also talk about different platforms. Chapter 4 documents all the experiments performed to demonstrate the stack overflow issues and analyze eBPF composite stack issues. Chapter 5 talks about design considerations from the analysis made and provides a stack switch solution. Chapter 6 compares our work with existing industry standard approaches and we summarize our work. In Chapter 7 we explore possible limitations on the proposed solutions. Also, we identify some discussion points and improvements that can be made in order to further develop the concerns of my research.

Chapter 2

Background

In this section we provide background on the Linux safe kernel extension architecture i.e, eBPF and its components that can affect the system stack memory.

2.1 Extended Berkeley Packet Filtering

The shift towards extensible software architectures necessitates the implementation of secure, efficient, and user-friendly extension mechanisms. Dynamic extension mechanisms are prevalent across various software domains, including web browsers [7], IDEs [42], media players [23], and even operating systems [13, 21, 31]. These mechanisms allow for the addition of third-party code to software, enhancing it with additional features and functionalities dynamically at runtime. Notably, operating systems have traditionally been an ideal environment for implementing extensible software observability, security, and networking functionality on-demand due to the kernel's privileged ability to oversee and control the entire system [14]. Hence, using dynamic kernel extension mechanisms, the kernel avoids recompiling the whole kernel source tree and rebooting it every time to add or remove code.

In Linux, traditionally, kernel extensions are implemented through Linux Loadable Kernel Modules (LKM). These modules are typically used for adding new driver functionality, new filesystem drivers, etc. However, it can be considered unsafe, as a buggy module can crash the kernel[38]. However, Extended Berkeley Packet Filtering (eBPF) has revolutionized this

landscape by enabling safe and efficient kernel extensions inside Linux. The original conventional Berkeley Packet Filtering (BPF) primarily served network packet filtering purposes [3] which, after doing simple checks, interprets user provided simple BPF byte code program. The introduction of eBPF marked a significant advancement bringing increased expressiveness, performance and safety with its verifier and runtime [41]. This ushered in a new era of tooling, empowering developers to swiftly diagnose issues, innovate, and enhance operating system functionality in real-time, all within the kernel itself [45]. Importantly, this can be achieved without necessitating alterations to the kernel source code or loading kernel modules. Beyond Linux, eBPF's influence extends; its successful porting to Windows has rendered it an industry-wide technology with wide-ranging applicability [40].

2.2 eBPF Architecture

Fig 2.1 depicts an overview of the eBPF architecture. BPF, together with its instruction set, offers several load time components like its verifier and JIT compiler, and runtime components like maps serving as efficient key/value stores, helper functions enabling interaction with and utilization of kernel functionality, tail calls to invoke other BPF programs, security hardening primitives, and attachment infrastructure within the kernel [4]. eBPF programs operate on an event-driven basis, executing when the kernel triggers hook points located throughout the kernel or user application.

2.2.1 eBPF load time

An eBPF program can be loaded into the Linux kernel by calling the `bpf()` system call with user provided eBPF byte code program. At load time, to guarantee the secure execution of user-space programs within the kernel, *the eBPF verifier* conducts thorough static checks on the BPF bytecode. These checks are designed to prevent various potential issues, such as

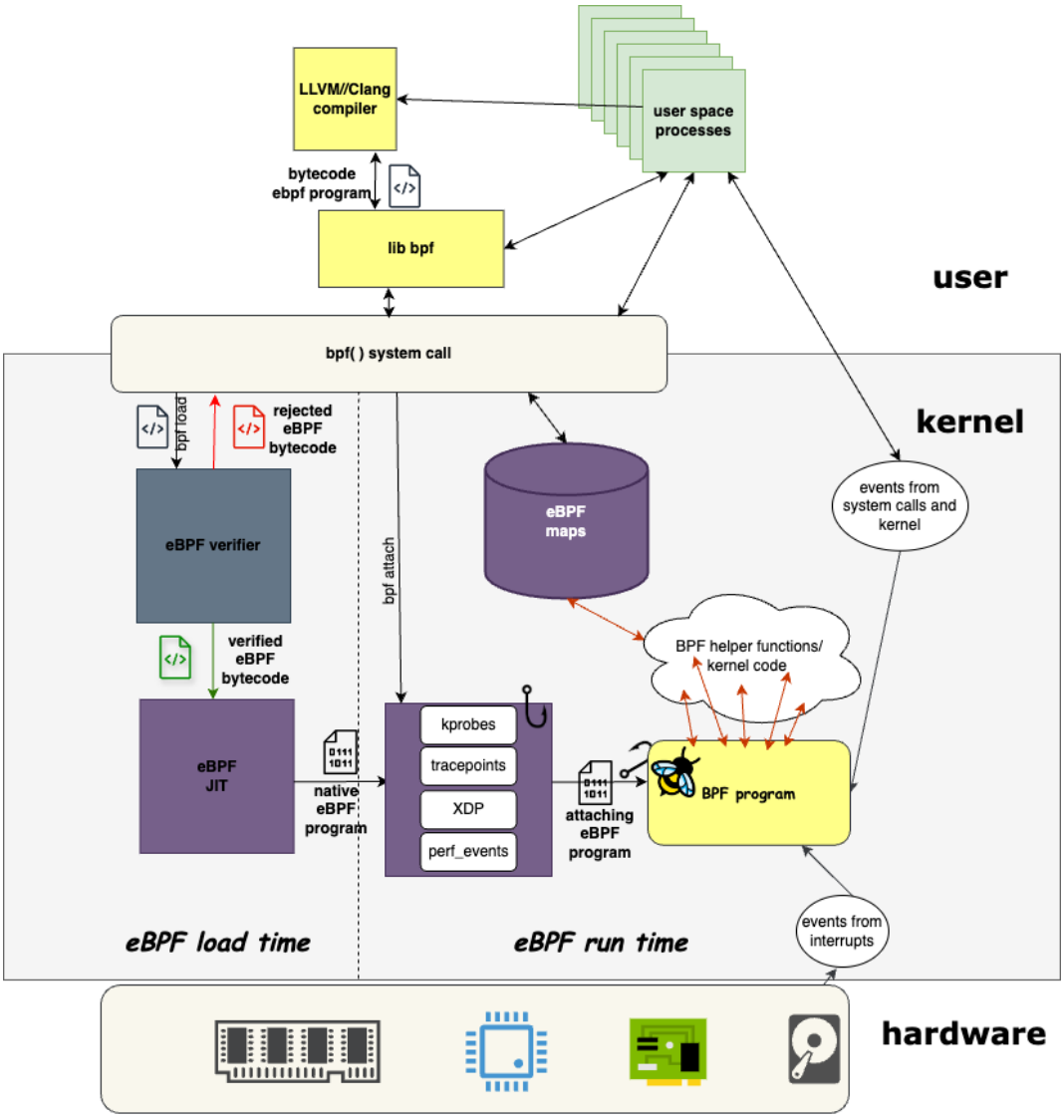


Figure 2.1: eBPF architecture

programs that run indefinitely (unbounded execution), make unexpected jumps or branches (invalid control flow), access memory beyond their designated areas (unauthorized memory access), expose sensitive kernel data, or attempt functionalities without the necessary permissions (capability violations).

Once the eBPF bytecode passes the safety checks from the verifier it is forwarded to *the in-kernel JIT compiler* which further compiles the bytecode into native machine code to speed

up its performance. Optionally, if JIT is disabled then an eBPF interpreter (not shown) will do the execution of the programs decoding the bytecodes on the fly.

Currently, at load time the BPF subsystem imposes a strict limit on each eBPF program's stack memory to 512 bytes only. However, these limits change to 256 bytes when the BPF program has the combination bpf2bpf and bpf tailcalls, which are eBPF runtime components and will be further discussed in the next section.

2.2.2 eBPF runtime

BPF Attachments: For enabling the already loaded eBPF program user initiates attach using the same `bpf()` system call which attaches at the desired hook point. Upon any event reaching these hook points, the CPU triggers and executes the corresponding BPF program. eBPF utilizes various mechanisms to attach programs at different points within the kernel or application. Hook points can be predefined like `tracepoints` or for scenarios lacking such predefined points, mechanisms like `kprobes` or `fentry` allow for program attachment at arbitrary locations within the kernel.

BPF Helper functions: Any running eBPF program can enhance the expressiveness by directly calling bpf helper functions, which are a set of stable well-defined and unverified kernel functions. These functions offer eBPF programs to interact with the kernel environment and provide various functionalities to access kernel code, kernel data structures. Similarly, eBPF also can use BPF Kernel Functions or more commonly known as kfuncs which lack a stable interface but are exposed for use by BPF programs. The verifier here makes the boundary where it only verifies the user provided untrusted code and trusts any kernel code from the helper function. We can consider this an escape hatch: any unsafe event happening inside the helper functions or kfuncs can affect the composite safety [18].

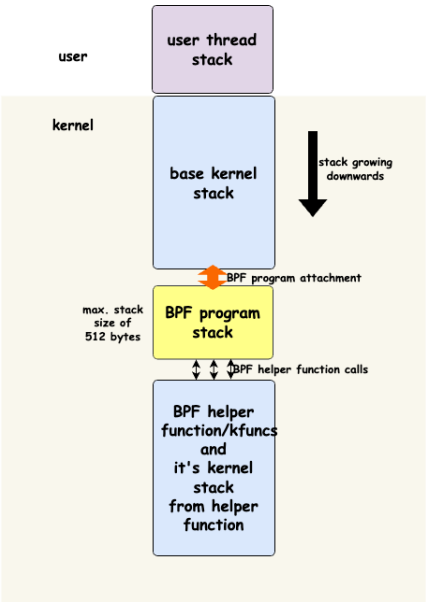


Figure 2.2: eBPF stack layout

When a verified BPF program executes from an attachment point, it typically inherits the current kernel stack. In the similar way, any bpf helper function calls made from the eBPF program also uses the same kernel stack as shown in Fig 2.2.

BPF tail calls and bpf to bpf calls: At runtime, a BPF program (caller) can directly call another BPF program (callee) to run next using the BPF tail call helper function. These tailcalled BPF programs actually initiate a long jump from the caller BPF program, reusing the current stack frame of the caller's program and permanently diverting control flow without returning. BPF tail calls provide a method for breaking down complex programs into smaller dynamic chains of programs; moreover, they reduce overheads associated with calling conventions by enabling simple jumps and reusing the latest stack frame Fig.2.3.

Moreover, inside individual BPF programs, standard function calls are also implemented as BPF-to-BPF calls, where each called function, or subprogram, possesses its own stack frame, similar to traditional functions. The main entry function in the eBPF program

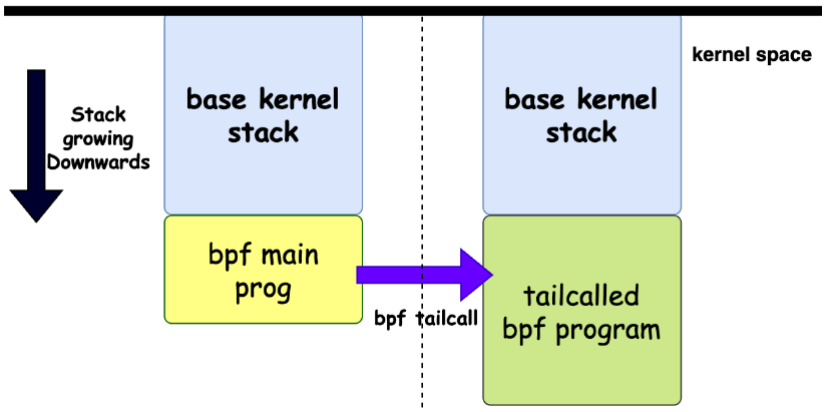
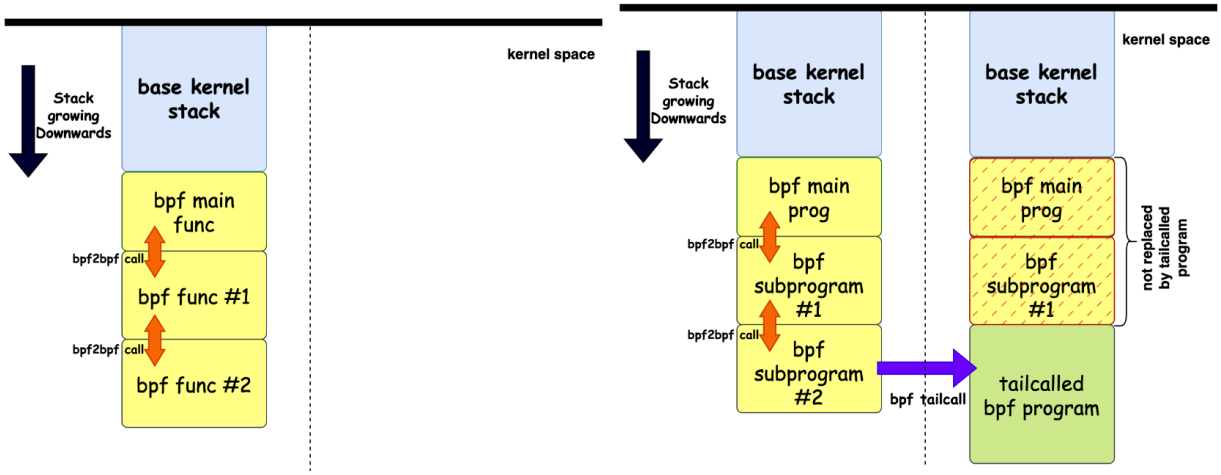


Figure 2.3: Stack state illustration of BPF program calling another program using BPF tailcall functionality



(a) bpf2bpf calls

(b) bpf tail call made from a subfuction called using bpf2bpf calls

Figure 2.4: Using BPF tail calls and bpf to bpf calls

code is called the program and all other BPF subprograms can be nested like functions in user-space programs. At load time, the verifier makes sure that, for any individual eBPF program, these BPF-to-BPF calls are capped to a maximum of 8 calls and also don't exceed the allowed 512 bytes to prevent potential stack overflow issues.

In our thesis we explore these helper functions and its kernel call graph from the perspective of control flow using stack memory and provide insights of how helper functions can be better equipped providing the safety guarantee that BPF subsystem provides to the system.

Combining bpf2bpf calls and bpf tail calls: Mixing BPF-to-BPF function calls with BPF tail calls is allowed where some BPF program can be tailcalled from another BPF callee's subprogram instead of the main/entry program. In this case, the tailcalled BPF program reuses the subprogram's stack frame and not the below one(s) of the BPF callee program like shown in Fig. 2.4. At BPF loadtime the checks are made when the combination of tail calls with BPF-to-BPF function calls are being used i.e, by reducing the maximum allowed stack size per BPF program from 512 bytes to 256 bytes. Furthermore, the maxmium number of BPF tail calls is also limited to 33 in total. In chapter 4.2.1, we demonstrate of how we can leverage the bpf2bpf/tailcalls combination maximum limits to derive an 8K stack size.

Chapter 3

Review of Literature

3.1 Safe Runtime Composition of Dynamic Extensions

The ability to dynamically add and remove software extensions from untrusted sources is increasingly desired across various software systems, which includes network architectures [24], web servers [44], graphics softwares [46], and database systems [19, 27] for obtaining better performance and improved functionality. In the domain of operating systems extensions there is a large literature around architectures trying to implement extensions which can be coarsely classified as better microkernels [9, 20, 21], exokernels [13], virtual machines [2, 39], and downloading untrusted code into the kernel [3, 26, 28, 29, 31].

In our thesis, we concentrate on the monolitic kernel class of extensible mechanisms, specifically downloading untrusted code into the kernel. Various extension mechanisms are using Type-safe languages [29, 31] or Interpretive languages [3], Software fault isolation [28] and Hardware fault isolation [26]. Previous efforts to build extensible systems using these have demonstrated the three-way tension between extensibility, safety and performance [31] mainly from the perspective of extensions can be untrusted and the kernel execution environment is trusted and safe to run the extension.

3.2 Measuring stack usage

Correctly measuring the maximum stack memory usage from a function is crucial for obtaining a true estimation of the stack needed to identify potential stack overflow conditions during composite stack executions. Another domain which requires precise stack usage estimation of its applications is real-time embedded systems, especially those in safety-critical roles. Also, resource constraints in these systems demand early defect detection to avoid costly fatal bugs, which are difficult to fix after deployment. Overprovisioning for unforeseen needs is impractical due to limitations in cost, size, and power[17, 25, 30]. Generally, for these problems, many solutions propose to minimise the stack usage by optimising using various techniques like using different scheduling policies [17] or using event-driven architectures using one stack [11, 12] instead of multi-threaded architecture.

One approach is to dynamically estimating the stack usage measurement by stack pollution check techniques [32], involve setting up the stack memory region to a specific value and then identifying the *polluted* area (memory locations without the initial value) after test execution. For instance, some RTOSes like FreeRTOS employ a runtime analysis technique called *stack painting* [15]. In Linux, this requires thorough testing like using fuzzing tools [6, 36] or using respective test suites for those modules. However, this approach provides a post-execution measurement of stack usage and is highly dependent on the specific test paths that trigger stack pollution which may not be completely covered or in some unknown exception paths so are not suitable for extension use cases.

Another technique is to statically analyse the control flow graph and determine the maximum stack usage [33, 34, 35]. These static stack size analysis analyse complete source/binary code like by using compiler tools [1]. Hence, unlike a dynamic approach this gives a very wide coverage of the paths including all exeception paths and also all untested paths. However,

static approaches suffer from overestimated measurement of stack usage, any recursions in the path, indirect functions calls, also any dynamically added code which is not analysed.

A representative paper exploring algorithms for measuring and predicting runtime stack usage is *Compiler-Assisted Maximum Stack Usage Measurement Technique for Efficient Multithreading in Memory-Limited Embedded Systems* [22]. One algorithm focuses on recording the maximum stack usage observed during program execution. Another algorithm aims to predict and prevent runtime stack overflows. This prediction is achieved by inserting code that estimates the stack usage based on current usage and the function's known stack footprint. However, for performance-critical kernels like Linux and extensions like eBPF, such frequent code insertions could be something to measure. In Chapter 5, we propose a hybrid approach. We utilized static analysis to determine the maximum depth of helper function calls which may not be as big as the Linux kernel. Instead of inserting checks in every function, which would overload the kernel, we leverage the static analysis results and implement a single, more efficient check (further details will be provided in the Solutions chapter).

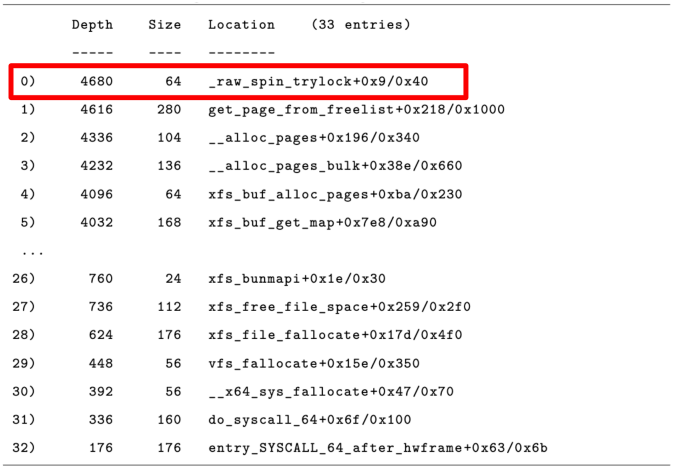
Chapter 4

Methodology

Our methodology section explores practical techniques and demonstrates eBPF stack composition safety issues, which can lead to unforeseen kernel stack space exhaustion during eBPF execution. To illustrate the stack safety issue in this section, we first discuss our approach for identifying deep stack attachment points in the Linux kernel. Next, we delve into our method for generating significant stack memory consumption using eBPF program(s), and finally, we combine these approaches to overflow the composite kernel stack due to eBPF execution.

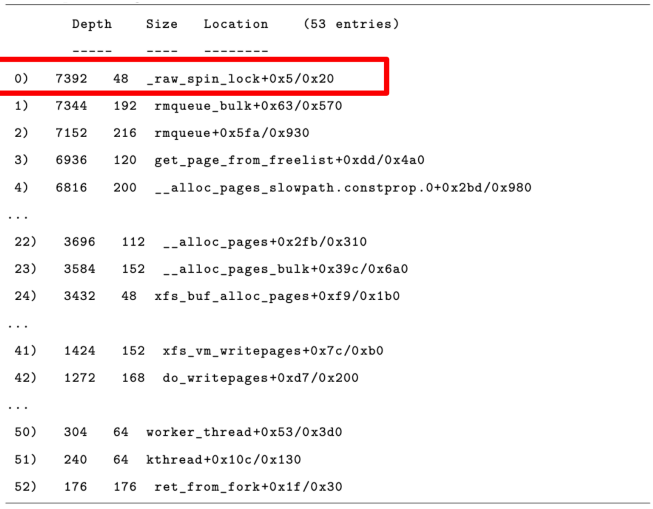
4.1 Finding deep stack attachment points within the Linux kernel

To discover the deep stack state for dynamic attachments, we will explore how we force specific Linux kernel components under certain scenarios to grow very deep kernel stacks. Our approach utilizes dynamic analysis by selecting a stack-intensive Linux subsystem and executing it in a constrained environment. We monitor stack usage with the ftrace kernel tool. We also considered alternative approaches like using a fuzzing tool such as Syzkaller and statically analyzing the Control Flow Graph (CFG) of all Linux kernel entry points. However, fuzzing tends to result in shallower stacks[16], and static analysis of the entire kernel can be unreliable and overwhelming to produce meaningful results.



	Depth	Size	Location (33 entries)
0)	4680	64	_raw_spin_trylock+0x9/0x40
1)	4616	280	get_page_from_freelist+0x218/0x1000
2)	4336	104	__alloc_pages+0x196/0x340
3)	4232	136	__alloc_pages_bulk+0x38e/0x660
4)	4096	64	xfs_buf_alloc_pages+0xba/0x230
5)	4032	168	xfs_buf_get_map+0x7e8/0xa90
...			
26)	760	24	xfs_bunmapi+0x1e/0x30
27)	736	112	xfs_free_file_space+0x259/0x2f0
28)	624	176	xfs_file_fallocate+0x17d/0x4f0
29)	448	56	vfs_fallocate+0x15e/0x350
30)	392	56	__x64_sys_fallocate+0x47/0x70
31)	336	160	do_syscall_64+0x6f/0x100
32)	176	176	entry_SYSCALL_64_after_hwframe+0x63/0x6b

(a) XFS test running in normal conditions



	Depth	Size	Location (53 entries)
0)	7392	48	_raw_spin_lock+0x5/0x20
1)	7344	192	rmqueue_bulk+0x63/0x570
2)	7152	216	rmqueue+0x5fa/0x930
3)	6936	120	get_page_from_freelist+0xdd/0x4a0
4)	6816	200	__alloc_pages_slowpath.constprop.0+0x2bd/0x980
...			
22)	3696	112	__alloc_pages+0x2fb/0x310
23)	3584	152	__alloc_pages_bulk+0x39c/0x6a0
24)	3432	48	xfs_buf_alloc_pages+0xf9/0x1b0
...			
41)	1424	152	xfs_vm_writepages+0x7c/0xb0
42)	1272	168	do_writepages+0xd7/0x200
...			
50)	304	64	worker_thread+0x53/0x3d0
51)	240	64	kthread+0x10c/0x130
52)	176	176	ret_from_fork+0x1f/0x30

(b) XFS test running under memory constrained conditions

Figure 4.1: ftrace logs of XFS deeper stack attachment points

Linux subsystems like file and networking subsystems are long known for using large stacks [43] as their intricate operations often demand deeper stacks. We leverage XFS filesystem and employ its regression test suite xfstest to trigger various scenarios. For reference, the system configuration was an Intel x86_64 architecture running in a VM with 1GB memory and 1GB swap space. When monitered using ftrace it gives stack usage of around 4KiB as shown in 4.1.

To achieve the maximum attachment depths, the code execution path ideally needs a large number of function calls. This typically occurs in systems executing under exceptional paths, such as low memory conditions, where the kernel works extra hard to manage user processes. To simulate these conditions, we run xfstests in a constrained environment with limited memory. Techniques like demand paging for the xfstests will encourage the kernel stack to grow deeper. Simulating scenarios where memory availability might be low, we rerun the test with the system memory now configured with 258KB memory and 2GB swap space. The maximum observed kernel stack usage was approximately 7-8KB, as shown in

Listing 4.1. On x86_64 architectures, Linux threads in kernel space are typically allocated 16KB stacks. This means the available memory for such a thread would be less than 9KiB after accounting for the stack (For x86_32 architectures, the kernel stack size is even smaller at 8KB).

Analysis of ftrace logs reveals that the kernel stack often contains layers of stack frames resulting from memory management functions searching for free memory, synchronization primitives such as spin locks, and scheduling functions. For monitoring or for debugging if any BPF extension is dynamically attached using mechanisms like Kprobes or Fentry requiring more stack space than the remaining stack memory would constitute an unclean attachment, potentially leading to stack overflows causing kernel crashes or memory corruption or compromising the system.

4.2 Creating sizable stack consumption using eBPF

To create a large stack using eBPF we need to understand possible BPF stack space requirements. The runtime components that are involved in ebpf using stack memory during runtime are bpf helper functions, ebpf tail/bpf2bpf calls, and the bpf maximum allowed stack space by the verifier. In our approach we first understand each BPF stack component and augment the stack size to its limits determining the unknown stack space requirements of composite BPF program execution.

4.2.1 Implementing sizeable chain of BPF programs

The current eBPF design allows programmers to use up to 33 nested bpf_tail_call instructions if interleaved with bpf2bpf calls, with each BPF program having a maximum stack space of 256 bytes. To achieve a substantial eBPF program size, a chain of 33 tail calls can

be implemented, transitioning from one BPF program to another. Each program could then consume the maximum stack space (256 bytes). The implementation logic for this approach is illustrated in Figure 4.2 and the stack trace in Listing 4.1. Each program contains at least two bpf2bpf calls and a tail call function invoked from the final function. The preceding BPF functions within the program utilize most of the available stack space, leaving minimal room for the final stack frame. This final frame simply calls another BPF program using the tail call helper, effectively replacing the small top-level BPF function stack. Through this approach, a chain of 33 BPF programs, each consuming approximately 256 bytes, results in a total program size of roughly 8,448 bytes. Additionally, the existence and sizes of these tail-called BPF programs are unknown at runtime, as they are dynamically invoked from the BPF maps by another BPF program, so this size cannot be computed statically.

Listing 4.1: Sizable nested eBPF programs. Tail call programs stack trace from stack depth 808 to 8712 bytes

```
1         Depth    Size   Location    (53 entries)
2         -----    ----   --------
3   0)     8832      48   bpf_bprintf_prepare+0x9/0x720
4   1)     8784      72   bpf_trace_printk+0x61/0x100
5   2)     8712     232   bpf_prog_2cf2d_testing_bpf_to_bpf_calls32+0xc3/0x117
6   3)     8480     232   bpf_prog_f4697_testing_tail_func32+0x119/0x11e
7   4)     8248     248   bpf_prog_2cf98_testing_tail_func31+0x145/0x14c
8      ...
9  34)      808     248   bpf_prog_1d3de_testing_tail_func+0x140/0x147
10     ...
```

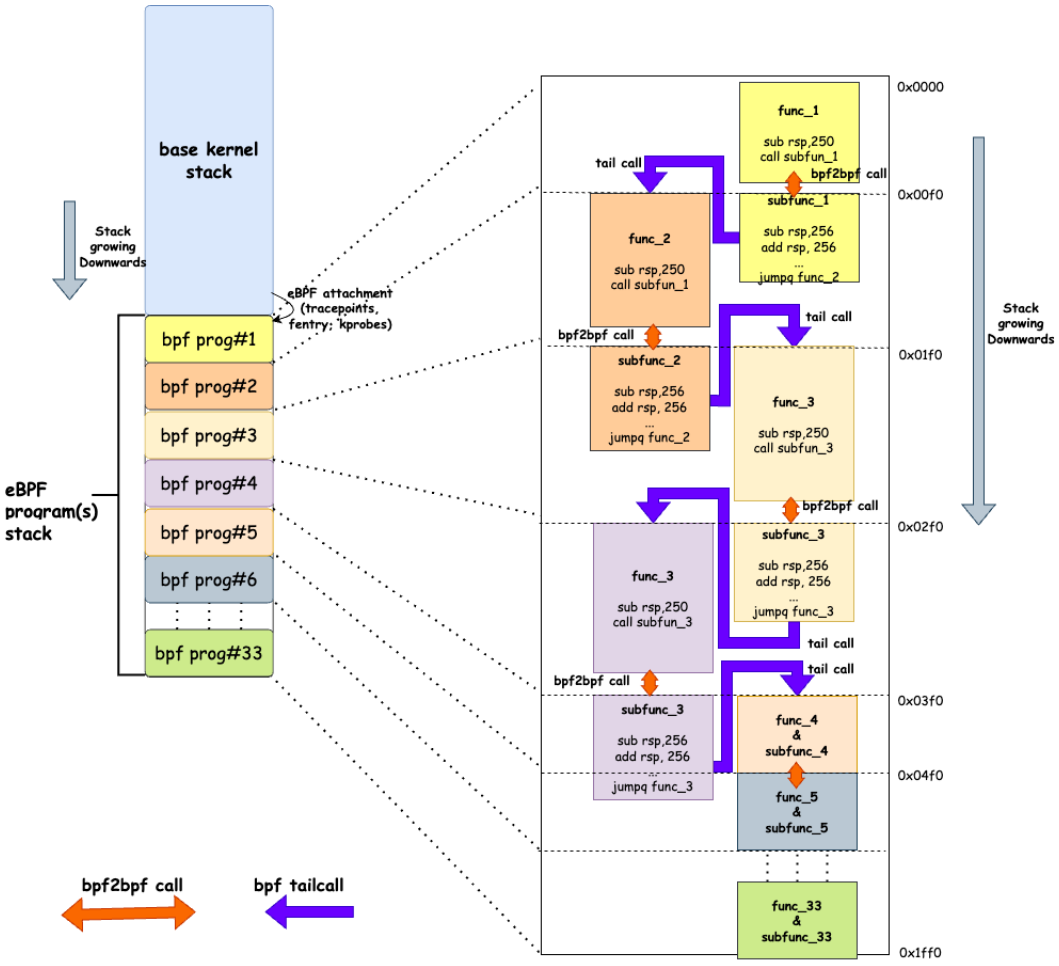


Figure 4.2: eBPF nested tail calls

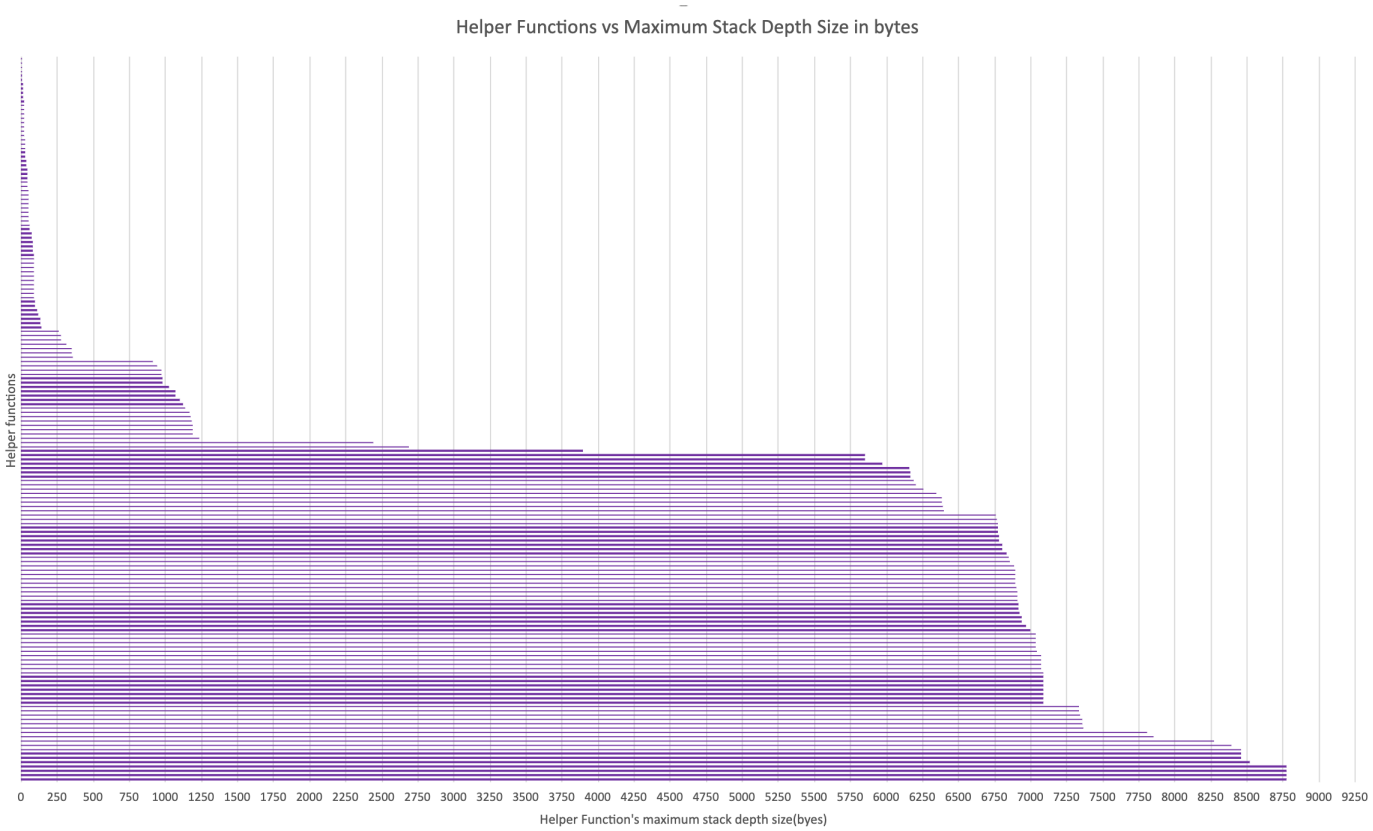
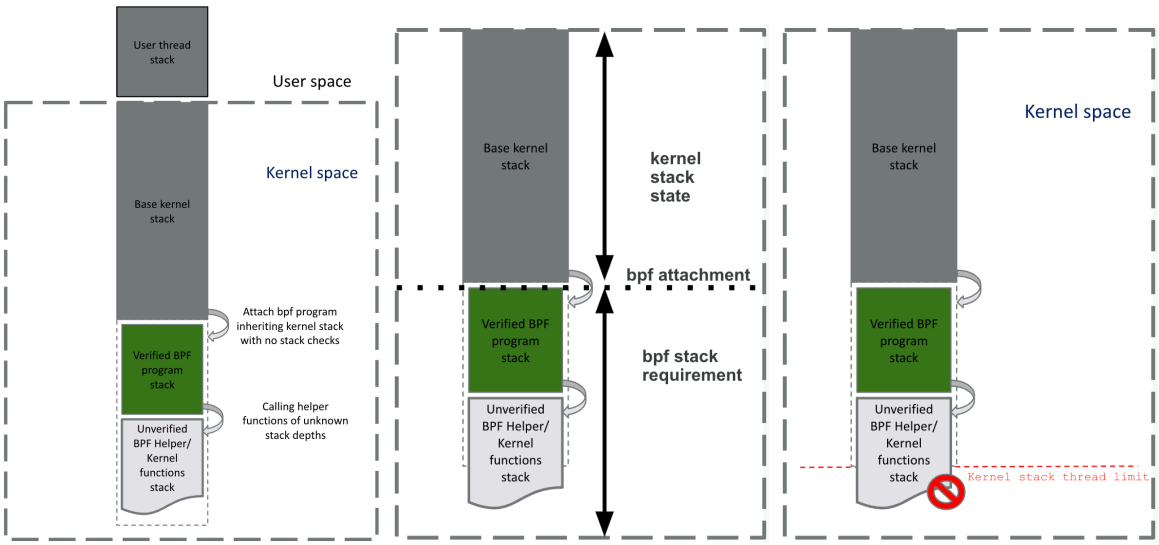


Figure 4.3: eBPF helper functions stack depths

4.2.2 Growing deeper eBPF helper function depths

Analyzing the control flow of the unverifed helper functions and the underlying kernel functions they involve is crucial to explore potential stack usage. We employ static analysis approach to hint the deep helper stack depths by generating call graphs which can reveal potential control flow explosions that lead to the creation of deeper kernel stack frames, ultimately exceeding the anticipated stack space.

For this purpose, we utilize the LLVM toolchain and processes all kernel LLVM bitcode files, generating a comprehensive set of modules of all kernel functions and their direct calls. These modules are used to create an adjacency list, starting from all BPF helper function calls. This process results in a comprehensive helper call graph that can be analyzed for



(a) BPF stack layout (b) BPF composite stack issues (c) BPF composite stackoverflow

Figure 4.4: Implications of unsafe eBPF stack composition

potential stack depths. Figure 4.3 illustrates the maximum call depths identified for all helper functions. Our static analysis currently has limitations. We only consider the direct call cost for all helper functions. The path analysis does not account for indirect calls, or back edges within the call graph, but since we are only using this to provide hints as to deep helper stack depths it is sufficient. Based on the data collected from helper functions stack depths, we selected the costly functions like bpf_get_stack() which are invoked from the final BPF program within the chain, effectively adding to a sizable stack space consumption.

Therefore, here the verifier relies on two critical assumptions about the kernel's runtime to restrict the depth of verified BPF programs [47]:

1. Kernel stack will always have 8 KB of stack space available for a BPF program to run.

2. The total size of the BPF program's kernel stack and the stack for any helper functions it calls will be less than 8 KB.

By combining the aforementioned techniques of growing a very deep linux kernel stack of around 7KB, attaching a nested tailcalled BPF programs of 8KB and finally calling a costly helper function of around 7KB size we were able to trigger a stack overflow within the Linux kernel. In our test environment, guard pages (enabled by default) detected the overflow and triggered a kernel crash. The associated crash log is presented in Figure 4.4. This should never have happened, as the BPF execution verifier guarantees kernel safety and is intended to protect the system from any bugs.

In next chapter, we will propose a design based on the static analysis conducted earlier, which will ensure that the kernel always has sufficient stack space available for a BPF program and its helper functions.

Chapter 5

Design and Implementation

This chapter addresses the eBPF runtime stack composition issues identified with the kernel stack. We begin by exploring design considerations for better awareness of the starting offset on the kernel stack and for conservative stack size estimation. Leveraging these estimates, we propose two solutions to accommodate the increased stack requirements: stack switching and stack extension. Later sections will evaluate these solutions and discuss their limitations.

5.1 Estimating per eBPF program maximum stack size requirements

To estimate the total stack space needed for BPF execution, we take a conservative approach. This considers both the BPF program itself and the worst-case scenario for helper function calls. This generic estimate applies to all eBPF programs, regardless of the specific helper functions used.

Through static analysis, we identify the deepest helper function which becomes our assumed maximum stack depth for any control flow triggered by helper functions. Additionally, the BPF program's stack space itself is straightforward, requiring a total of almost 8KB to accommodate the maximum BPF program chain as calculated previously. By combining these estimates, we arrive at a conservative total stack size for BPF execution which can be

validated experimentally.

In both approaches, we allocate stack memory for BPF execution in units of pages. This means we round the conservative total BPF stack size up to the nearest whole page size. For example, in our current case, the combined worst-case requirements are 6KB for `bpf_get_stackid()` and 8KB for the maximum BPF chain, totaling 14KB. On x86_64 systems, this would be rounded up to 4 pages, resulting in a new allocation of 16KB of clean stack space.

While building the Linux kernel, the statically analyzed data of the maximum depths of every BPF helper function is added as a header file in the kernel source directory: `include/linux/bpf_helper_max_depth.h`. This header file contains a `bpf_helpers_max_depth` array, listing all BPF helper functions in the format of `BPF_FUNC_<helper_function>`, indexing the helper offset in the `bpf_max_helper_depth[]` array and assigning its respective maximum stack depth value calculated (shown in fig.5.1). Additionally, this header file also contains the value of the deepest among all available helper functions in the kernel, stored in the macro `BPF_HELPER_DEEPEST_STACK_SIZE`.

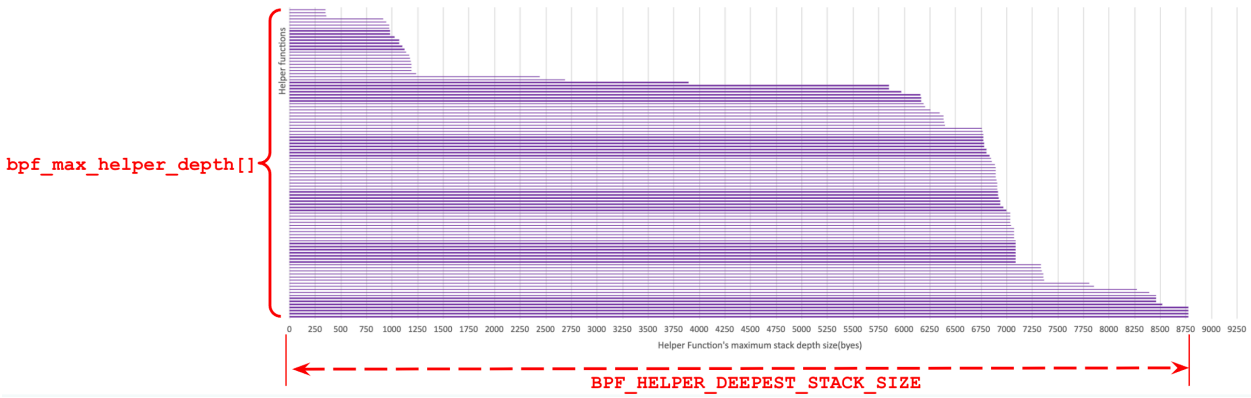


Figure 5.1: Estimating stack size requirements

Listing 5.1: Example `bpf_helper_max_depth.h` header file format in the kernel source tree which will be used by the verifer later

```
int bpf_max_helper_depth[] = {
        [BPF_FUNC_sys_bpf] = 7624,
        [BPF_FUNC_bind] = 7504,
        [BPF_FUNC_copy_from_user_task] = 7088,
        ...
        [__BPF_FUNC_MAX_ID] = -1
};
```

5.2 Solution 1 - Kernel Stack Switch

To ensure the safe execution of eBPF programs within the kernel (composite safety), a clean stack is a prerequisite whenever control flow enters BPF program. Our first approach to solving this problem involves switching the stack from the current, potentially unclean, kernel stack to a newly allocated *emergency stack*. This emergency stack will be sized according to the previously estimated maximum BPF stack size.

This switching logic is implemented at the verifier and at the attachment point, which varies depending on the specific mechanism (e.g., tracepoints, kprobes, fentry) used to attach the eBPF program. Currently, support is available for these three attachment types.

When loading the BPF program, the verifier stores the total BPF stack requirements in the `stack_thr` field of the `struct bpf_prog`. Essentially, the verifier checks two conditions. Firstly, if there is any combination of bpf2bpf calls and tail calls being used, in which case the `stack_thr` size is set to 32*256 bytes; otherwise, it is set to 512 bytes. Secondly, the verifier, upon encountering each helper call instruction, reads the respective maximum stack

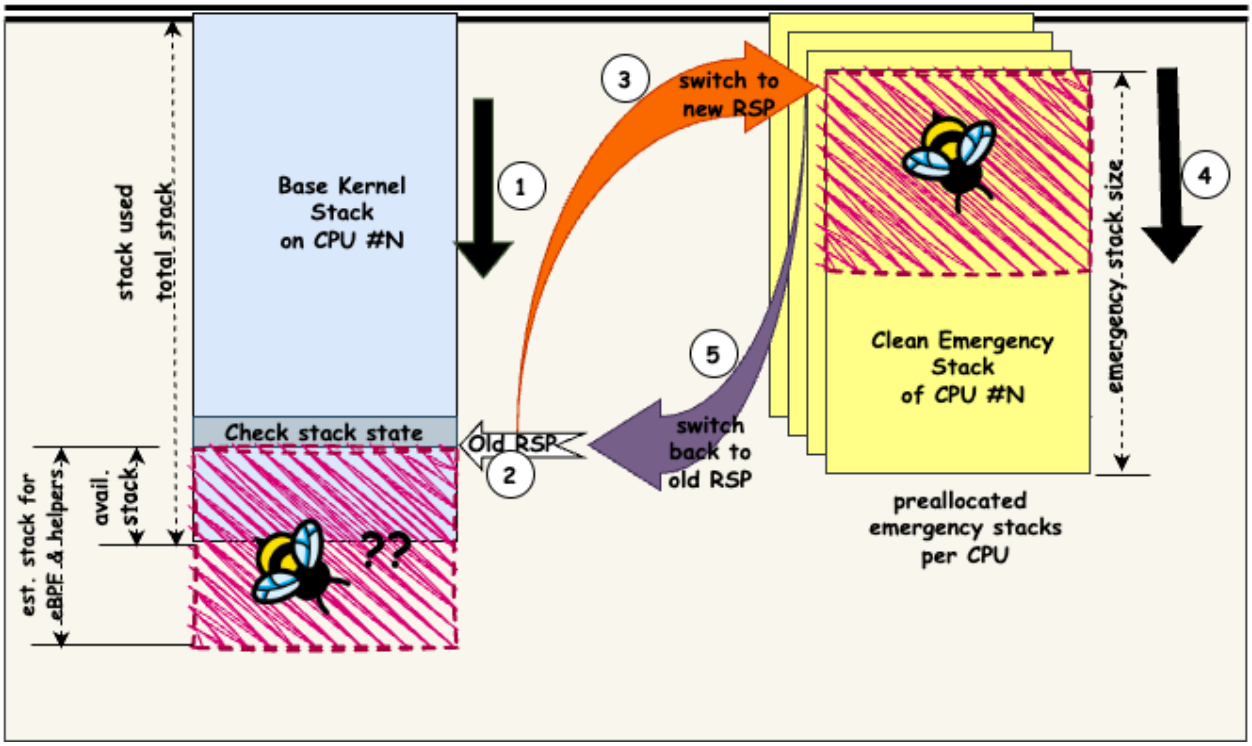


Figure 5.2: Check stack state and switch stack if needed.

depth from the `bpf_max_helper_depth[]` array and stores the maximum value of all helper functions used in that BPF program. This maximum helper function's depth is then added to the `stack_thr` field, providing a conservative estimate of stack usage size.

While attaching the logic works by first *checking* the available memory on the kernel stack using as shown in Listing 5.2. If the remaining space falls below a threshold defined by `stack_thr` field in `struct bpf_prog` the stack pointer is updated to point to the top of the pre-allocated emergency stack resulting in a *stack switch*. The program utilizes the emergency stack for all subsequent calls, including bpf2bpf calls, bpf tail calls, helper function calls, and even kernel function calls. Calls will push data onto this clean stack memory, and returns will unwind the stack frames as expected. Finally, upon returning from the BPF program's base function, control flow seamlessly jumps back to the original kernel stack, restoring the previous stack context. If there is sufficient memory then no switching is done.

Listing 5.2: Example `bpf_helper_max_depth.h` header file format in the kernel source tree which will be used by the verifer later

```
// Stack Overflow Check
if (in_interrupt() || ((u64)kernel_stack - (u64)end_of_stack(current) > prog->
    stack_thr)) {
          return bpf_func(ctx, insnsi);
}
// Swap rsp to emergency stack and call bpf program
```

Emergency BPF stacks are pre-allocated on a per-CPU basis during the kernel boot-up phase of size `BPF_HELPER_DEEPEST_STACK_SIZE` bytes ceiled to the nearest page boundary. The BPF subsystem is designed to disable preemption and context switching by default whenever a program starts execution. This ensures the program finishes without interruption. Consequently, our design is simplified as we only need one emergency stack per CPU.

Listing 5.3: Emergency BPF stacks of size i.e, deepest of all helpers stack depth size + (no. of chained bpf progs) x (per bpf prog max stack) bytes ceiled to the nearest page boundary

```
// Size of Emergency stack memory
#define BPF_EMR_STACK_SIZE (((BPF_HELPER_DEEPEST_STACK_SIZE + 33*256 + 4095)
    >> 12) << 12)
```

5.3 Solution 2 - Kernel Stack Extension

Another simple solution is to increase the current default stack size by `BPF_HELPER_DEEPEST_STACK_SIZE` more bytes for all threads. This way we can also suppress the minimal stack switching costs entirely. Below illustrates the code modifications required for extending the stack size.

A potential drawback of this approach lies in the memory overhead associated with virtual memory allocation (vmalloc). For each thread, the kernel needs to locate virtually contiguous pages for the composite kernel stack. These pages are physically allocated immediately during thread creation (fork). This can become a significant performance bottleneck in two scenarios: systems with limited memory running a large number of threads, where finding contiguous virtual space becomes challenging; and situations with bursts of short-lived threads, where frequent vmalloc operations can negatively impact performance. Additionally, the kernel's performance might suffer if there's insufficient free memory, as the fork process would require extra time to locate and acquire suitable memory pages, mirroring the performance issues identified in the analysis section. However, this approach remains attractive when memory limitations are less of a concern and performance improvement is the primary goal.

```
1 // Change THREAD_SIZE macro for extending the stack which should be in
     multiples of PAGE_SIZE.
2
3 #define THREAD_SIZE  ((PAGE_SIZE << THREAD_SIZE_ORDER) + BPF_EXT_THREAD_SIZE)
```

5.4 Limitations

It's important to acknowledge that our proposed approach, like the current static method, focuses solely on direct helper function calls. It does not currently account for indirect calls, loops, or nested programs. These factors can also influence stack usage and will require further investigation to develop a more comprehensive solution for dynamic stack size estimation.

Kernel Modifications: Kernel functions might be implemented dynamically by kernel modules or vary depending on the source files used during compilation. This makes it difficult to

predetermine and calculate the maximum helper function call depth for all scenarios.

Nested BPF Programs: As mentioned earlier, the stack switching approach currently doesn't handle nested BPF programs and ignores them when encountered. The stack extension approach also inherits this limitation.

Indirect Calls: The current design does not consider indirect calls, which are prevalent in the Linux kernel. These calls can potentially lead to unexpected control flow paths and increase stack usage.

Recursion and Loops: The current design also ignores recursions and loops within the CFG path. These scenarios can significantly increase stack usage and need to be addressed in future work.

Reliability of stack traces: Another challenge with the stack switching solution is ensuring reliable stack traces, such as those used in ftrace. With two stacks in use, attempting to traverse the stack from the emergency stack may prove difficult due to differing starting points. It is generally expected that, even in the presence of bugs or kernel panics, the system must be capable of traversing the stack reliably.

Chapter 6

Evaluation

In this chapter, we discuss our experimental setup and results, focusing on functionality, memory and latencies of the stack switching and stack resizing solutions. Our final experiments are conducted on a Virtual Machine which uses the QEMU hypervisor equipped with 4-core CPU and 2GiB memory using Linux v6.7 kernel. We perform micro benchmarks for both the solutions and a macro benchmark using XRP for the stack switching solution.

6.1 Extended Stack Solution

For the stack increase solution, we evaluate and compare the executions of BPF programs with default base kernel stack configuration (i.e. 16KB or 4 pages) and the extended kernel stack configuration extended by BPF_EXT_THREAD_SIZE bytes (16KB or 4 pages) to the default stack size. We run the same XFS test benchmark test used in the methodology chapter here to prove the effectiveness of the solution. In the ftrace stack dump of List 6.1 we can see the extended kernel stack case where it safely accommodates the BPF execution which was previously overflowing in the default base kernel stack configuration. Also, the two execution stack sizes and it's compositions are plotted in Fig. 6.1 graph where it compares the BPF execution insufficiency on default stack are many available kernel stack space with the extended stack solution.

Listing 6.1: Stack trace of BPF execution on an extended kernel stack from base kernel stack of 16KiB to 32KiB

```
        Depth    Size   Location    (97 entries)
        -----    ----   --------
  0)    17608      48   desc_read_finalized_seq+0x5/0xe0
  1)    17560     152   _prb_read_valid+0x90/0x320
  2)    17408      16   prb_read_valid+0x1b/0x30
  3)    17392     216   printk_get_next_message+0x73/0x2a0
  4)    17176     128   console_flush_all+0xf4/0x400
  5)    17048      56   console_unlock+0x7e/0x100
  6)    16992      64   vprintk_emit+0x164/0x1f0
  7)    16928      96   _printk+0x5c/0x80
  8)    16832     104   get_perf_callchain+0x54/0x1d0
  9)    16728      64   bpf_get_stackid+0x85/0xd0
 10)    16664      16   bpf_get_stackid_raw_tp+0x77/0xa0
 11)    16648     144   bpf_prog_f08fbf4_testing_bpf_to_bpf_calls33+0xd7/0xdc
 12)    16504     144   bpf_prog_66fbebf_testing_tail_func33+0xc8/0xcf
 13)    16360     152   bpf_prog_b74983f_testing_tail_func32+0xea/0xf1
 14)    16208     152   bpf_prog_635772c_testing_tail_func31+0xea/0xf1
    ...
    ...
```

The major downside of extended kernel stack solution is fragmentation. As the kernel stack space pages are virtually contiguous and physically allocated while forking from it's parent thread this means that every thread spawned can have lots of unused available memory most of the thread's lifetime. It can be of a big concern when there are flood of thread spawns being made demanding memory. Every fork() will also experience the overhead due to vmalloc allocations.

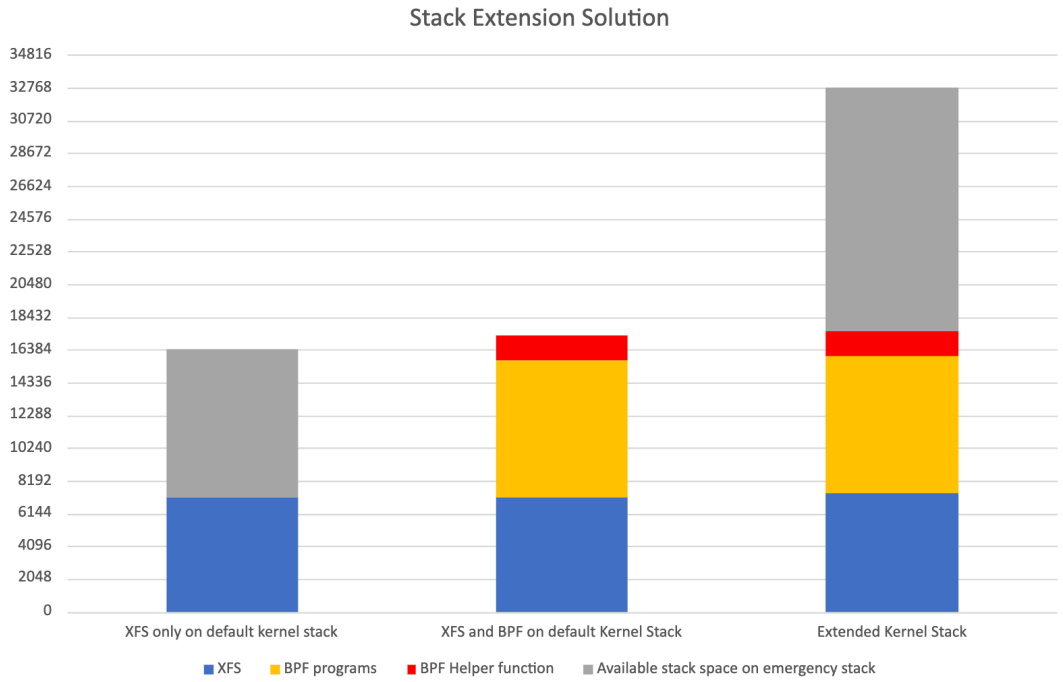


Figure 6.1: Stack extension showing the XFS only execution with BPF execution on default unclean kernel stack and clean extended kernel stack. Here x86_64 stack limit is currently 16KiB

6.2 Stack Switch Solution

For the stack switch solution, first we evaluate the working of the solution likewise done for the previous extended stack solution. If a stack overflow condition is detected we execute on the clean emergency stack, however the stack trace using ftrace cannot be obtained on the emergency stack which is one of those limitations of our switching solution. The simple way to know the unknown helper function's depth is by doing a simple 1 page extension to view the stack trace which is similar to the List. 6.1 observed in the stack increase solution. The comparison of stack sizes between single default base kernel stack case vs dual stacks using base kernel stack and emergency stack can be seen in the graph 6.2.

The runtime costs of the stack switching solution is tabulated in 6.1. On every trigger regardless of switching costs every instance has to pay the stack overflow check condition, also

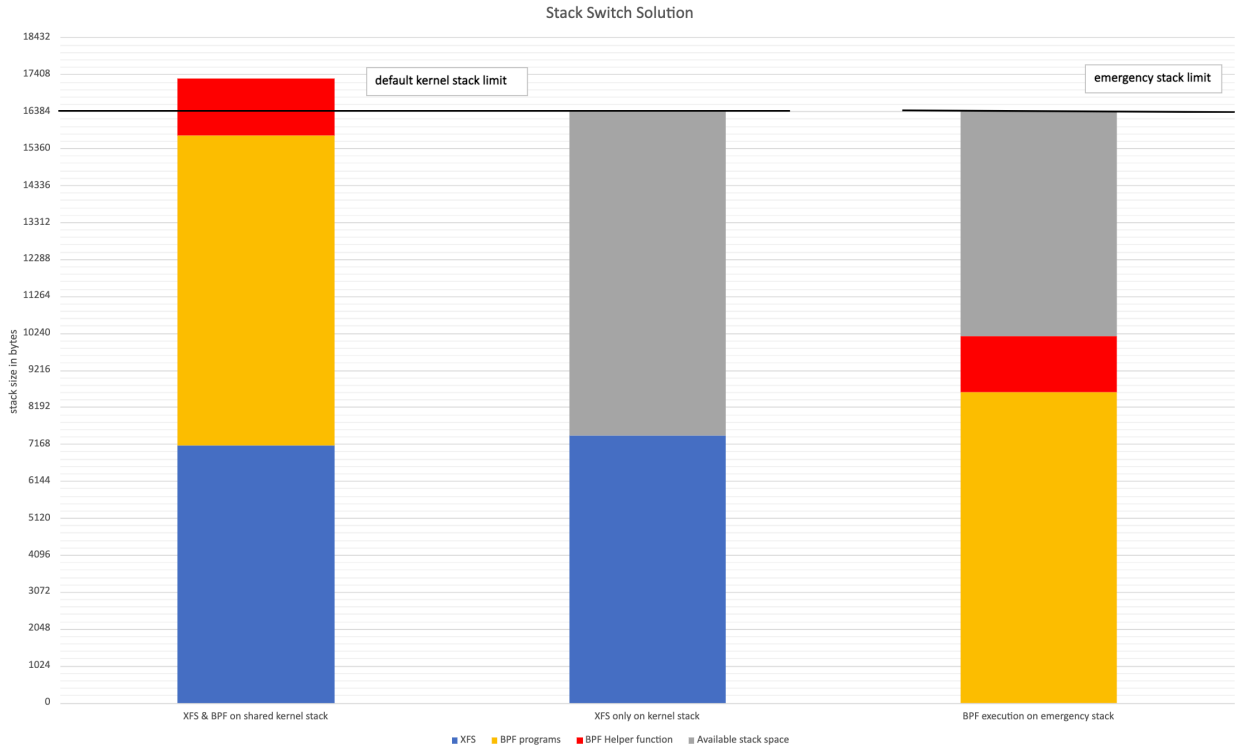


Figure 6.2: Stack Switching Solution:

Stack overflow check	Stack overflow check and Switching cost	One Time emergency stack creation cost
11ns	20ns	792ns

Table 6.1: Stack Switch solution runtime cost

the bootup can also be a bit slower now as it has to find free pages for the per CPU emergency stack, and finally the switching cost of the BPF program to and from the emergency stack if there is a stack overflow situation. An empty tracepoint BPF program takes 200ns which is 5 percent overhead for each invocation.

Chapter 7

Discussion

In this section, we'll discuss the key findings of our research, how it might impact other fields, and potential areas for future development.

Our research highlights the importance of having a holistic perspective on the BPF subsystem within the kernel. In certain cases, the composition of the BPF subsystem presents fundamental safety challenges, particularly in ensuring stack safety within eBPF programs. The central focus is on how eBPF programs use specific features such as helper calls, tail calls, attachment points, indirect pointers, and recursion, which may introduce uncertainties in safety assessments. These uncertainties shed light on inherent weaknesses in the BPF subsystem's ability to fully ensure program safety at runtime.

7.1 Nesting of eBPF programs

eBPF programs can be nested, where a program calls both a helper function where another BPF program is attached to that helper function or a kernel function within the helper's control flow. Similar to prior runtime issues with attachments, tail calls, and helper functions, the verifier remains unaware of this nesting, leaving it outside its purview. However, since these extensions fall under the BPF subsystem's responsibility, safety guarantees should still hold true.

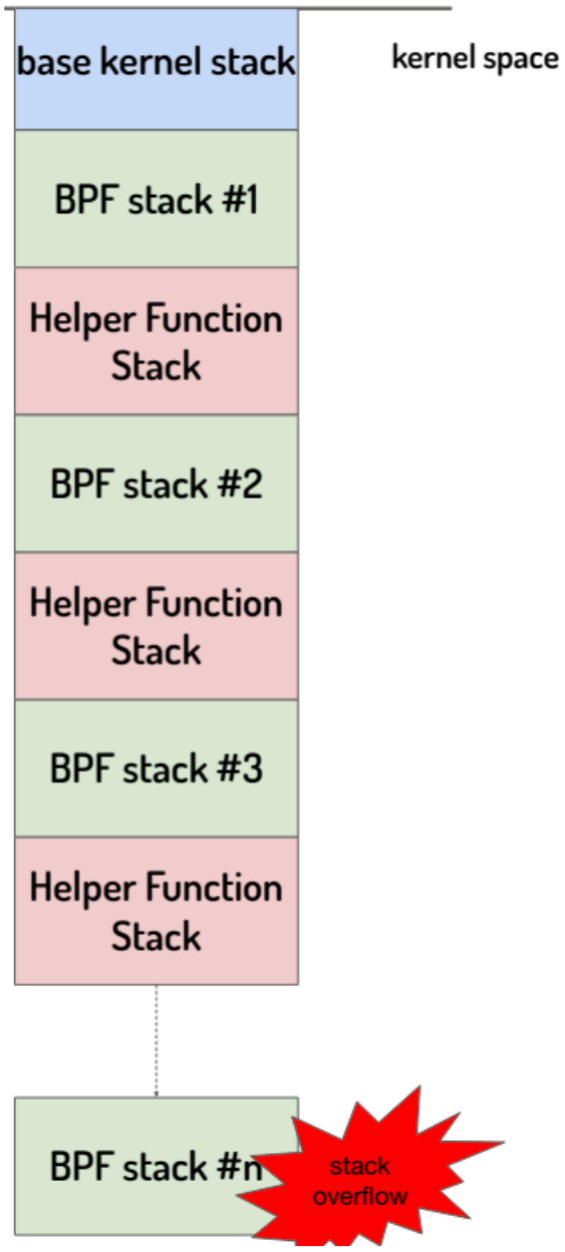


Figure 7.1: Nesting of eBPF programs in helper paths

A particular nesting scenario becomes problematic when the nested program calls the same helper function that triggered its execution. This can lead to an infinite loop, as both programs consume the same stack space upon inheritance, potentially causing a system crash. Fortunately, this specific issue is mitigated by the fact that nesting the same program on the same CPU is disallowed for kprobes, tracepoints, and fentry.

However, eBPF intentionally allows nesting different multiple programs within fentry (unlike tracepoints or kprobes). This enables multiple programs to run on the same CPU using fentry, which could lead to stack overflow if not managed carefully. This raises an interesting avenue for further research of exploring how to enable the execution of multiple nested BPF programs while ensuring safety constraints are met.

7.2 Unsoundness of static estimate for maximum helper stack depth

The Linux kernel's core design utilizes indirect pointers and function pointers to achieve object-oriented-like functionalities with C, exemplified by structures like fops. These constructs present a challenge for the static analysis of eBPF programs within helper functions.

The primary issue lies in indirect function calls within the helper function's control flow path. Static analysis struggles when dealing with unknown indirect pointers, as they could theoretically target any kernel function. This makes it nearly impossible [37] to determine the exact stack space required for such calls.

A similar problem arises when the control flow originating from a helper function contains loops, recursions, or cycles that lead back to the same helper function. These loops introduce uncertainties in stack usage analysis using static methods, making it difficult to predict the

required stack space accurately.

This highlights the need for innovative approaches to enhance static analysis capabilities in eBPF. Finding methods to handle these unknown paths and accurately determine stack usage is crucial for ensuring safety and preventing potential issues like stack overflows.

7.3 BPF tail calls

The fundamental problem with tail calls is that it unwinds the current stack frame of the subprogram being called, but does not remove the stack frames of all preceding subprograms. This results in *stack space gaps* and an overall increase in BPF stack usage on the same kernel thread. This problem becomes even more relevant for composable programs where tail calls occur dynamically at runtime.

Further research is warranted to explore potential mitigation strategies. Ideally, we should investigate methods to allow the tail-called BPF program to reuse the leftover memory from previous stack frames, potentially improving memory efficiency and reducing overall stack usage.

Chapter 8

Summary

In summary, our thesis explored the eBPF subsystem's fundamental problem in it's stack setup where the BPF execution is not aware of the composite stack start state while attaching and also the composite stack requirement of BPF execution. To prove this we devised a methodology of dynamically making the kernel stack unclean for attaching a verified BPF program. Further we statically analysed the BPF helper functions control flow graph to determine the maximum stack requirement to make a conservative decision of potential stack overflow conditions. We have proposed an intuitive design of switching stack if the attachment is deemed unclean from the remaining stack memory available compared to the max stack depth. Finally, we have evaluated our design with various micro and macro benchmarks.

Additionally, various aspects of future work that can be explored in this research are discussed where we explain the possible nesting scenarios in BPF helper paths, possible indirect calls and recursions altering the paths and the dynamic stack size estimation at load time for improved efficiency.

Bibliography

[1] Proving the absence of stack overflows. URL https://link.springer.com/chapter/10.1007/978-3-319-10506-2_14.

[2] Composing os extensions safely and efficiently with bascule. URL https://dl.acm.org/doi/abs/10.1145/2465351.2465375.

[3] The bsd packet filter: a new architecture for user-level packet capture. . URL https://dl.acm.org/doi/10.5555/1267303.1267305.

[4] Bpf architecture. https://docs.cilium.io/en/latest/bpf/architecture/, .

[5] ebpf.io. https://ebpf.io/, .

[6] Brf: ebpf runtime fuzzer. URL https://arxiv.org/abs/2305.08782.

[7] The research of plug-in extension technology based on android multimedia player platform. URL https://ieeexplore.ieee.org/document/5974152.

[8] Comparing security in ebpf and webassembly. URL https://dl.acm.org/doi/abs/10.1145/3609021.3609306.

[9] Curios: Improving reliability through operating system structure. URL https://www.usenix.org/legacy/event/osdi08/tech/full_papers/david/david_html/.

[10] Extended berkeley packet filter: An application perspective. URL https://ieeexplore.ieee.org/abstract/document/9968265.

[11] A dynamic operating system for sensor nodes. . URL https://www.usenix.org/conference/mobisys2005/dynamic-operating-system-sensor-nodes.

[12] System architecture directions for networked sensors. . URL https://pdos.csail.mit.edu/archive/6.097/readings/tinyos.pdf.

[13] Exokernel: an operating system architecture for application-level resource management. URL https://dl.acm.org/doi/abs/10.1145/224057.224076.

[14] ebpf: A new approach to cloud-native observability, networking and security for current (5g) and future mobile networks (6g and beyond). URL https://ieeexplore.ieee.org/abstract/document/10138542.

[15] Freertos, real-time operating system for microcontrollers. https://www.freertos.org/Stacks-and-stack-overflow-checking.html.

[16] Exploratory review of hybrid fuzzing for automated vulnerability detection. URL https://ieeexplore.ieee.org/abstract/document/9541397.

[17] Minimizing stack and communication memory usage in real-time embedded applications. URL https://dl.acm.org/doi/abs/10.1145/2632160.

[18] Kernel extension verification is untenable. URL https://dl.acm.org/doi/10.1145/3593856.3595892.

[19] An illustra technical white paper. URL https://dl.acm.org/doi/10.1145/381854.381899.

[20] L4 microkernels: The lessons from 20 years of research and deployment. URL https://dl.acm.org/doi/abs/10.1145/2893177.

[21] Mach: A system software kernel. URL https://www.sciencedirect.com/science/article/pii/0956052190900045.

[22] Compiler-assisted maximum stack usage measurement technique for efficient multi-threading in memory-limited embedded systems. URL https://link.springer.com/chapter/10.1007/978-3-642-21375-5_10.

[23] Empoweb: Empowering web applications with browser extensions. URL https://ieeexplore.ieee.org/document/8835286.

[24] A survey of active network research. URL https://ieeexplore.ieee.org/document/568214.

[25] Optimizing stack memory requirements for real-time embedded applications. URL https://ieeexplore.ieee.org/abstract/document/6489571.

[26] Integrating segmentation and paging protection for safe, efficient and transparent software extensions. URL https://dl.acm.org/doi/pdf/10.1145/319151.319161.

[27] The postgres next generation database management system. URL https://dl.acm.org/doi/10.1145/125223.125262.

[28] Efficient software-based fault isolation. URL https://dl.acm.org/doi/10.1145/168619.168635.

[29] Singularity: rethinking the software stack. URL https://dl.acm.org/doi/abs/10.1145/1243418.1243424.

[30] Eliminating stack overflow by abstract interpretation. URL https://dl.acm.org/doi/abs/10.1145/1113830.1113833.

[31] Spin: an extensible microkernel for application-specific operating system services. URL https://dl.acm.org/doi/abs/10.1145/504390.504408.

[32] Portable multithreading–the signal stack trick for user-space thread creation. . URL https://www.usenix.org/conference/2000-usenix-annual-technical-conference/portable-multithreading-signal-stack-trick-user.

[33] Eliminating stack overflow by abstract interpretation. . URL https://dl.acm.org/doi/10.1145/1113830.1113833.

[34] Static checking of interrupt-driven software. . URL https://dl.acm.org/doi/abs/10.5555/381473.381478.

[35] Stack size analysis for interrupt-driven programs. . URL https://dl.acm.org/doi/10.5555/1760267.1760276.

[36] syzkaller. https://github.com/google/syzkaller.

[37] Where does it go?: Refining indirect-call targets with multi-layer type analysis. URL https://dl.acm.org/doi/10.1145/3319535.3354244.

[38] Synthesizing safe and efficient kernel extensions for packet processing. URL https://dl.acm.org/doi/abs/10.1145/3452296.3472929.

[39] Virtuos: an operating system with kernel virtualization. URL https://dl.acm.org/doi/abs/10.1145/2517349.2522719.

[40] ebpf for windows. https://microsoft.github.io/ebpf-for-windows/.

[41] Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. URL https://dl.acm.org/doi/pdf/10.1145/3371038.

[42] Microsoft. extension api. https://code.visualstudio.com/api, 2021.

[43] Jonathan Corbet. Expanding the kernel stack. https://lwn.net/Articles/600644/.

[44] The Apache Software Foundation. The apache software foundation. https://www.apache.org/.

[45] Brendan Gregg. Linux extended bpf (ebpf) tracing tools. https://www.brendangregg.com/ebpf.html.

[46] J. Heid. "mastering adobe premiere 5" macworld. https://www.apache.org/.

[47] Siddharth Chintamaneni Sai Roop Somaraju and Dan Williams. Overflowing the kernel stack with bpf. In *Linux Plumbers Conference 2023*, November 2023. URL https://lpc.events/event/17/contributions/1595/.