# Re-thinking termination guarantee of eBPF

Raj Sahu

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
Godmar Back
Michael Le

May 07, 2024
Blacksburg, Virginia

Keywords: eBPF, performance prediction, Linux, termination

# Re-thinking termination guarantee of eBPF

Raj Sahu

(ABSTRACT)

In the rapidly evolving landscape of BPF as kernel extensions, where the industry is deploying an increasing count of simultaneously running BPF programs, the need for accounting BPF-induced overhead on latency-sensitive kernel functions is becoming critical. We also find that eBPF's termination guarantee is insufficient to protect systems from BPF programs running extraordinarily long due to compute-heavy operations and runtime factors such as contention. Operators lack a crucial mechanism to identify and avoid installing long-running BPF programs while also requiring a mechanism to abort such BPF programs when found to be adding high latency overhead on performance-critical kernel functions. In this work, we propose a runtime estimator and a dynamic termination mechanism to solve these two issues, respectively. We use a hybrid of static and dynamic analysis to provide a runtime range that we demonstrate to encompass the actual runtime of the BPF program. For safe BPF termination, we propose a short-circuiting approach to skip all costly operations and quickly reach completion. We evaluate the proposed solutions to find the obtained performance estimate as too broad, but when paired with the dynamic termination, can be used by a BPF Orchestrator to impose policies on the overhead due to BPF programs in a call path. The proposed dynamic termination solution has zero overhead on BPF programs for no-termination cases while having a verification overhead proportional to the number of helper calls in a BPF program. In the future, we aim to make BPF execution atomic to guarantee that kernel objects modified within a BPF program are always left in a consistent state in the event of program termination.

# Re-thinking termination guarantee of eBPF

Raj Sahu

(GENERAL AUDIENCE ABSTRACT)

The Linux kernel OS has a relatively recent feature called eBPF which allows adding new code into a running system without needing a system reboot. Due to the flexibility offered by eBPF, the technology is attracting widespread adoption for diverse use cases such as system health monitoring, security, accelerating programs, etc. In this work, we identify that eBPF programs have a non-negligible performance impact on a system which, in the extreme case, can cause Denial-of-Service attacks on the host machine despite going through all security checks enforced by eBPF. We propose a two-part solution: the eBPF runtime estimator and the Fast-Path termination mechanism. The runtime estimator aims to prevent the installation of eBPF programs that can cause a large performance impact, while the Fast-Path termination will act as a safety net for cases when the installed program unexpectedly runs longer. The overall solution will enable better management of eBPF programs concerning their performance impact and enforce strict bounds on the added latency. Potential future work includes factoring in the impacts other than performance in our solution such as inter-BPF interaction and designing easy-to-use knobs which an operator can easily tune to relax or constrain the side-effects of the eBPF programs installed in the system.

*To mom, dad, and my love, Anmol.*

# Acknowledgments

I want to express my gratitude to my advisor, Dr. Dan Williams, whose guidance has been instrumental in getting this project to where it is today. His trust in me and the process always motivated me to push further and has helped me truly appreciate the art of research. I am also grateful for having amazing lab mates, where new ideas were always welcomed and critiqued. Lastly, I want to express my heartfelt thanks to my parents and my partner, Anmol. Without their support, I could have never made it this far.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

eBPF is an emerging kernel extension framework gaining rapid adoption due to its safety properties. These properties are enforced by a static analyser called the *BPF verifier* which traverses a BPF program to ensure memory safety and termination. However, in our experiments, we find the termination guarantee to be weak and practically broken. We first elaborate the problem of weak termination guarantee in the next section, before proposing two complementing solutions: 1) a runtime estimator to predict performance and prevent installation of long-running BPF programs; 2) a safe termination mechanism to provide a safety kill-switch which can safely abort a BPF program in case a program unexpectedly exhibits long-runtime and needs to be uninstalled. We then conclude by summarizing the contributions made in this thesis.

## 1.1 BPF programs with high runtime

While the BPF verifier guarantees that a BPF program will always terminate, it does not guarantee a quick execution which is often an unsaid assumption for BPF programs. A BPF

program can run with a high execution time due to runtime factors which could stem from either kernel-associated operations or due to other BPF programs existing in the system. There currently exists no reliable mechanism to determine how long-running these BPF programs can be or how they will affect the system in the worst case. A slow-running BPF program can prove to be a bottleneck at any frequently used call path within a system [33]. Thus, we assert that having a generic performance prediction of eBPF programs is now an indispensable requirement for system administrators who want to track performance penalties due to BPF. In chapter §2, we further investigate the impact of long-running BPF programs on a system through some examples and describe the requirement of dealing with such high-latency kernel extensions. In chapter §3, we explore different factors in the runtime environment that could lead to a BPF program executing much longer than it usually should, and then propose a runtime estimator for BPF programs. The runtime estimator will provide data points based on which an operator can make decisions about allowing the given amount of overhead into the system. To further deal with cases where an operator wants to uninstall an ill-performant BPF program, we propose a dynamic mechanism to terminate long-running BPF programs in chapter §4.

### 1.1.1 Runtime Estimation of BPF programs

Estimating performance impact of a BPF program faces the challenge of accounting for runtime overhead due to different branches of a program along with the BPF *helper* interface. The helper interface is an abstract wrapper around existing kernel functions which the BPF verifier trusts to never break any of its safety properties. Dynamic performance measurement faces the issue of completeness while static analysis of helper functions is non-tractable due to the large call-graph of kernel code. Additionally, some helper functions modify control-flow of BPF programs in a non-trivial way which further complicates static-analysis. The key

insight is that by utilizing the static analysis performed by the BPF verifier and performing dynamic offline measurements of helper functions, we can override limitation of using both the techniques individually. With the above insight, we propose a runtime estimator in chapter §3 that generates an estimate of the best-case to worst-case runtime of any BPF program. The proposed estimator takes inspiration from performance prediction of Network Functions (NFs) [28, 29]. We test the proposed solution against some sample eBPF programs from the Linux kernel source tree and observe the estimates to be aligned to actual runtimes in chapter §5.

We also discuss a potential use-case where the proposed runtime estimator can be integrated with a userspace BPF orchestrator for instrumenting policies that will keep a check on added latencies on critical data paths. By integrating with the proposed runtime estimator, an orchestrator can better estimate how much, in the worst-case, a call path gets affected if the given set of BPF programs are getting installed. This will also enable a new class of policies that will restrict the low-privileged users from adding too much latency on a critical hook point in the kernel. For example, added latency on a networking fast path could be tolerable under the order of a few nanoseconds, while for tracing-based applications it could be in the order of a few microseconds. We discuss a new direction of work along the lines of making better and easy-to-use policies in chapter §7.

### 1.1.2 Runtime Termination of BPF programs

While the above introduced Runtime Estimator will prevent BPF programs with bad performance from being installed into the system, programs with dynamic factors, which are only known at runtime, such as function arguments for helpers, iterator count for a loop, etc leads to an extremely broad runtime estimate which won't be helpful for an operator

for admission control. The operator in that case will only realize during execution that the running BPF program is taking more than acceptable resources, and needs to be terminated to protect other tasks in the system from starvation.

Termination of a kernel task faces the challenge of correctly releasing acquired resources such as locks which, if left unreleased, can lead to deadlocks in the system. With no assistance from the programming language (C for Linux kernel), bookkeeping resources for eBPF will greatly compromise performance specially for latency-sensitive applications like XDP[27]. There are some evolving techniques where the concept of safe points and exception-based synchronous-termination are being applied to BPF programs[22]. However, such approaches are not efficient for supporting abrupt program termination where the list of possible program points where termination can be requested is very large and spans the whole program.

Our key insight is that the BPF verifier, by ensuring that all resources are always released before a program exits, embeds the resource management in a BPF program's control flow. To support termination of a BPF program, we leverage this property to short-circuit a heavy-weight BPF program with a lightweight version that is guaranteed to always complete faster then the original BPF program. We describe the proposed Fast-Path termination mechanism in chapter §4 and evaluate the correctness and overheads in chapter §5.

Introducing termination to BPF, either synchronous or abrupt, breaks the unsaid assumption of BPF programmers that an executing BPF invocation always runs to completion. Aborting a BPF program, while is ensured to not leak memory or leave kernel locks unreleased, still can leave other kernel objects in an inconsistent state due to the early exit of the BPF program. We discuss this new direction of work necessitated by the termination mechanisms for BPF in chapter §7.

## 1.2 Contributions

To summarize the contributions, in this thesis we:

- Identify BPF termination guarantee to be weak and not sufficient to protect a system from high-latency BPF programs which can monopolize the CPU and starve other task in the system for an undesirable time frame.
- Propose a runtime estimator for BPF programs that provides necessary data to implement policies that can prevent long-runtime extensions from getting installed into the system.
- For long-running programs depending on dynamic parameters not known at load time, we propose a BPF termination mechanism which provides the flexibility to kill extensions on-demand while assuring safety.
- Evaluate the proposed Runtime Estimator and the Fast-Path termination mechanism for usability, correctness and different overheads.
- Identify future work which are enabled or necessitated due to the runtime estimator and the termination mechanism.

With the help of the two proposed components, a cloud server operator will further ascertain the safety of the system in the process of using BPF programs.

In the next chapter, we first give a background on the BPF subsystem of the Linux kernel, describe a use case where an operator would want to instrument policies to limit BPF program installation based on performance penalty, and then establish the case of long-running BPF programs and why we need BPF termination. Chapter §3 describes the design of the proposed Runtime Estimator and chapter §4 proposes the Fast-Path termination

approach. In chapter §5, we evaluate both the proposed sub-components for correctness, efficiency, and overheads. We then discuss some related work in the literature in chapter §6 followed by potential future work for this project in chapter §7.

# Chapter 2

# Motivation

In this chapter we deep dive into the issue of long-running BPF programs which we claim to make the existing BPF termination guarantee inadequate. We first provide a background on the existing BPF subsystem and define some of the system constraints which make the issue of long-running programs unique compared to any existing system. We also define the threat model under which this work has been pursued. We then introduce the role of BPF orchestrators in systems with increasing applications for BPF extensions and identify a potential gap where a runtime estimator can perfectly fit. For BPF programs with performance depending on runtime factors, we motivate the need for runtime termination using an example long-running program which throttles the CPU performance as per our evaluation and currently has no safe mechanism to prematurely abort. Together, the runtime estimator and the termination mechanism will allow a BPF orchestrator to enforce runtime based policies covering cases where a BPF program can either be prevented from install, and if not, then terminated when found to violate the runtime policies. Towards the end, we conclude the chapter with the key takeaways.

## 2.1 Berkeley Packet Filter

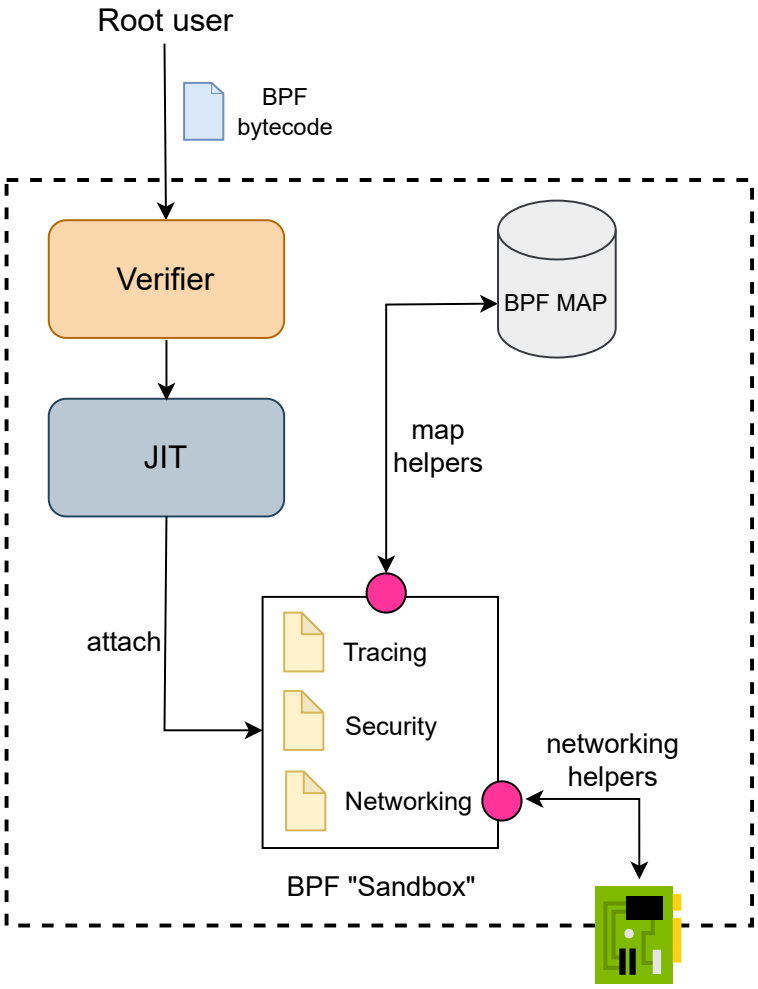


Figure 2.1: BPF subsystem uses helper interface to limit access to core kernel functions and objects

BPF is a extension framework for adding new kernel code at runtime and while providing safety properties for the installed extension. BPF extensions are becoming crucial to add dynamic features to a running system without requiring full system reboot and are currently being deployed for observability[25], security enforcement[3, 20] such as system call filtering[30], network firewall[27], load balancing[51], etc. While BPF was originally introduces as a much simpler fast-packet filtering technology, it got revamped to *eBPF* (formerly

extended Berkeley Packet Filter) while the older version is referred to as *cBPF* (classic BPF). For the sake of this thesis, we synonymously use BPF and eBPF.

eBPF depends on a static analyzer, known as BPF verifier, to ensure some safety properties are upheld for any installed BPF program which guarantee that an installed extension cannot crash the kernel in any way. Because of these safety properties, cloud service providers are now deploying hundreds of eBPF programs in their system to perform critical operations on-demand[24].

Whenever a new extension is installed, the verifier goes through all possible program paths of an extension to check for different classes of logical bugs that could crash the kernel. Currently, two classes of bugs are checked for: invalid memory operations, and back-edges in the BPF program that could lead to infinite looping. While having back-edges is usually not categorised as a bug, the eBPF design calls it so to prevent a program from running infinitely long and never terminating.

Figure 2.1 shows an overview of how the BPF susbsystem works in the Linux Kernel. The Linux kernel provides a system call to load and hook BPF programs to kernel functions. Once verified, the verifier makes several optimizations such as inlining *map* access (which are used to provide BPF-to-BPF and BPF-to-userspace communication) and function calls, dead code elimination, before passing the modified BPF bytecode to the *Just-In-Time compiler* (JIT). The JIT converts the bytecode into machine code (say, x86) and performs further rounds of optimization. The final machine code is then attached to the relevant hook point within the kernel. After installation, this BPF program will be executed in kernel context whenever the control flow reaches the associated hook point. During program execution, a BPF program is allowed to access to a limited helpers interface. The helper interface is aimed to provide only the necessary kernel functionality which comprises of network packet manipulation, event logging, etc.

Due to this well-defined but limited interface along with the verifier's static analysis, BPF programs guarantee several safety properties which are unique to BPF in the Linux kernel:

- **Memory Safety:** The verifier checks for potential NULL-dereferences, validates pointer arithmetic to ensure a no kernel object is illegally accessed or modified by the BPF program.
- **Termination:** To protect the kernel from executing an indefinitely long running BPF program, the verifier performs a graph traversal on the overall BPF program structure to ensure it is a Directed-Acyclic Graph. This prevents any back-edges that could have potentially led to infinite loops or recursions. To support bounded loops, the subsystem provides additional helper functions. BPF also imposes limits on program size, number of function calls, nesting levels, etc which further restricts the complexity and therefore the runtime of a BPF program.
- **Resource Cleanup:** BPF programs can get kernel resources like acquiring spin locks, taking reference to data objects, etc. To ensure memory is not leaked or locks not freed, the verifier traverses all possible program paths that will be reachable at runtime and tracks all memory allocations and de-allocations. By maintaining a list of live resources at any point in the program, the verifier can reject any program containing a call-path with unreleased resources.
- **Others:** The BPF verifier also provides other safety mechanisms like Spectre mitigation where the verifier checks for possible Spectre gadgets that a malicious attacker might attempt to install through BPF programs. If Spectre mitigation is enabled, the verifier suitably inserts instructions to prevent poisoning of CPU branch predictors.

With these checks in place, the BPF subsystem provides a "sandboxed" execution environment for BPF programs. Before diving into the issue of long-running BPF programs, we

describe the intended threat model and some constraints under which the BPF subsystem operates within the Linux kernel which needs to be considered to better understand the issues with long-running BPF programs.

**Threat Model: Privileged BPF**

The Linux kernel supports privileged and unprivileged BPF mode, where BPF program installation is limited to root user or is open to any user, respectively. While the original BPF design and the verifier were aimed to facilitate unprivileged BPF installs, several security vulnerabilites have led to disabling the unprivileged mode by default. In this work, we follow the trend and focus on privileged BPF installation and execution. Thus, the threat model for this work is the system administrator and other root users who want to add new features like tracing to their servers but may unintentionally end up installing buggy code which, if not correctly blocked by the BPF verifier, can lead to unstable kernel or worse, kernel panics.

**System Constraints: No Premption**

BPF programs, in the non-real-time configuration of Linux kernel, runs with preemption disabled which inhibits the kernel scheduler from running any other starving task in the system no matter how unfair this arrangement is proving to be. For generic kernel code, however, this feature of disabling preemption is reserved for critical section code, say accessing per-CPU variables, and thus generally means the time for which preemption is disabled, is very small. Due to this, other tasks waiting for their fair share of CPU resources are not starved. The reason for BPF to disable preemption is because of its need to acquire RCU locks which currently by design require no-preemption. While this arrangement works fine for most BPF programs, a potentially long-running BPF program will end up monopolizing the CPU until completion as the scheduler won't be able to schedule any other task waiting to be executed.

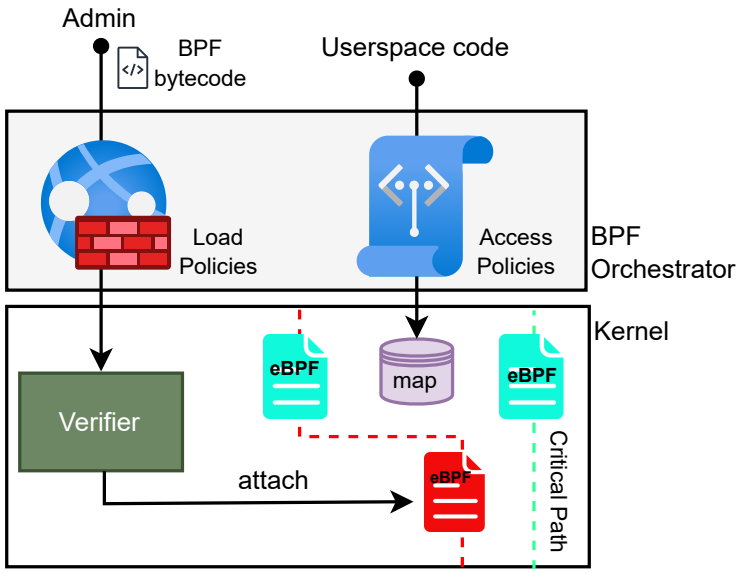


Figure 2.2: Regulating userspace interaction with BPF using RBAC. (Blue pages are optimized BPF programs, while red pages denote slower-running BPF programs which is slowing down one of the critical paths)

## 2.2 BPF Orchestration

As eBPF is finding use cases across diverse applications, BPF orchestrators are gaining importance as they provide efficient management of all the BPF programs in a system. Figure 2.2 gives an overview of how BPF orchestrators work in a cluster environment like Kubernetes. An administrator configures *load* and *access* policies. The load policies can include signature validation[17], restricting users to a limited set of system hook points their eBPF program can attach to, and to vary these policies for different pods to tighten security around operation-critical nodes. The access policies restrict the list of BPF programs a user can interact with using shared map objects. These policies can be per-user basis or role-based (RBAC) as per the size of the cluster.

When a user wants to load their BPF bytecode to a specific hook point, through the admin, the framework checks whether the user has enough permissions to attach to the requested

hook and then passes the BPF bytecode to the verifier. The verifier performs the static analysis and the framework, then, attaches the bytecode to the desired hook upon successful verification. Whenever a userspace program wants to access the map objects, it needs to pass through the access policies which authenticate the request. If permitted, the framework will provide APIs to read and write to the maps.

To summarize, the BPF-orchestrator, by enforcing policies, guards a system from multiple facets:

- Load policies prevent less-privileged users from loading BPF programs which can impact functionalities concerning performance and security.
- Access policies ensure safety from unwarranted reads/writes to BPF map objects which can affect execution or leak system information.

While the load and access policies provide fine-grained control over BPF programs and their interaction, the operator has virtually no insight into the runtime effects of these programs which introduces challenges. Tracking runtimes is a necessity for operators to meet SLA requirements in production environments. For example, users inadvertently hooking on critical paths like the network stack can lower the system's resilience against Denial-of-Service attacks. As the use of eBPF grows, multiple programs from multiple vendors on a critical path make it virtually impossible for an operator to reason about. Even though the existing policies prevent less-privileged users from compromising critical functionalities, they provide no control over the latency of BPF programs attached by the high-privileged operators. We further elaborate on the issue of high-latency BPF program using an example in section §2.2.1 and motivate the need of a runtime estimator which an operator or a BPF orchestrator can use to identify high-latency BPF programs and avoid installing them. In section §2.2.2, we motivate the need of a runtime solution which provides a kill-switch

```
1     int simple(void):
2         bpf_printk("foo")
3
4     int prog_n(void):
5         bpf_loop(1000,simple)
6             .
7             .
8             .
9     int prog_1(void):
10        bpf_loop(1000,prog_2)
11
12    int main():
13        bpf_loop(2000,prog_1)
14
15
```

Listing 2.1: Pseudo-code showing a sample long running BPF program using nested loops.

mechanism to deal with BPF programs which cannot be identified as high-latency during load-time in which the orchestrator can terminate BPF programs found to violate the set runtime policies. Finally in section §2.2.3, we tie up the two proposed solutions to picturize how a BPF Orchestrator will be able to enforce runtime-baed policies.

### 2.2.1 The need for BPF Runtime Estimates

The BPF verifier only guarantees eventual termination, which does not indicate how quickly will a program terminate. We wrote a simple eBPF program using the *bpf_loop* helper which can be made to run for several hours, which for any system is much more than needed to trigger an alarm. Listing 2.1 shows a simplified version of the code. As the only limit on levels of nesting is due to the limited stack size of BPF programs, adding more nested loop calls to the allowed maximum makes the runtime in the order of hours! Runtime estimation is therefore a critical requirement for better management of BPF-dependent systems. Such an estimate will not only flag low-performing programs but also will be useful to create policies that restrict highly unpredictable or very long-running programs from getting installed into the system.

```
1    int map_access(void):
2        k = bpf_get_prandom_u32()
3        v = bpf_get_prandom_u32()
4        bpf_map_update_elem(map_foo, k, v, flags)
5
6    int populate_map(void):
7        iter = bpf_probe_read_user(usr_ptr)
8        bpf_loop(iter, map_access)
9
10   int main():
11      populate_map()
12
```

Listing 2.2: Combination of resource-contending helpers and iterator helpers can give a long runtime

A naive approach based on dynamic benchmarking using *fuzzers* or the *bpftool test-run* feature faces the problem of incompleteness. An incomplete analysis can miss rare but costly branches which could eventually lead to unexpected worse-case runtimes. On the other hand, using static analysis to estimate runtime has historically faced the issue of state explosion due to function pointers and large call-graph. However, static analysis has not been investigated in the context of eBPF runtime.

We propose the BPF Runtime Estimator in chapter §3 which uses a hybrid of static and dynamic measurements to produce a best-to-worst case runtime range, during loading stage of the BPF program.

### 2.2.2 The need for BPF Termination

While the Runtime Estimator proposed above will prevent programs which can be determined to have a high-latency at load-time, many BPF programs will show performance varying dynamically due to factors like loop iteration count (for *bpf_loop*), length of a VMA list (for *bpf_find_vma*), etc. Similar to Listing 2.1, we create another BPF program (Listing 2.2) taking the *bpf_loop*'s iteration count dynamically from a userpace address using the

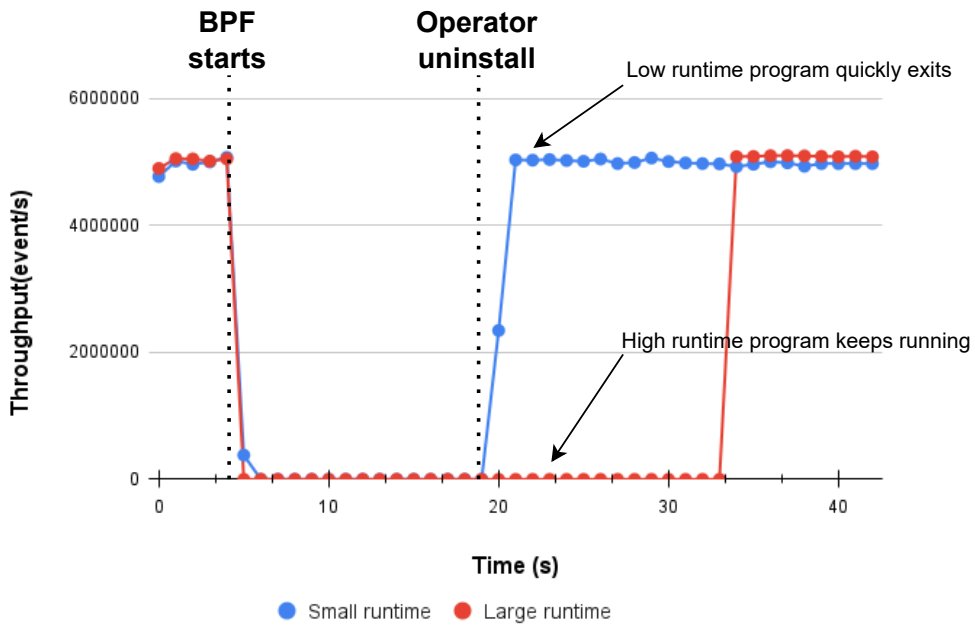


Figure 2.3: BPF programs with low runtime can be delinked to restore system performance. However for high runtime BPF programs the program will continue to execute even after uninstall is issued.

*bpf_probe_read_user* helper function to find the number of map entries to populate. Because of the iteration count being unknown at load time, this program's performance cannot be determined by the runtime estimator. To further amplify the resource utilization, we chose the *LRU_HASH* type BPF map which provides the Least-Recently-Used type map eviction to BPF programs. We benchmark this BPF program by attaching the program as a tracepoint to an empty system call and measure the number of calls made per second as the throughput. Figure 2.3 shows the obtained throughput measurement in red.

In the above figure, we observe an immediate performance throttling (0 calls/s) due to installation of the costly BPF program described in Listing 2.2. Trying to uninstall this BPF program, through the existing method of delinking it from the hook point, does not help regain the original system throughput as it does not interfere with an already executing program and the system has to wait until the program exits. Therefore, to get back control

over the stalled CPU, an operator either has to wait for the long-running BPF program to finish or if the wait is unacceptable, then reboot the system to gain back control over the CPU. For comparison, we install another relatively low resource-intensive BPF program, which only performs a 1000 map updates (fixed) compared to arbitrarily large number of updates in the original case. In this case (blue), while every BPF invocation is still costly enough to show a performance impact ($\approx$ 900 calls/s), the system performance restores about immediately after an operator decides to uninstall the BPF program. Thus, a termination mechanism is needed to abort long-running BPF programs without having to wait for the program to exit.

Termination of tasks running in kernel context is currently handled by disabling preemption when context switching or aborting a tast will be unsafe. However, as discussed in section §2.1, BPF programs run with preemption disabled which further worsens the issue of long-running BPF programs as they end up monopolizing the CPU and starve all other task in the CPU's run-queue. A trivial approach would be to issue an interrupt on the CPU running the target BPF program to skip rest of the execution, which, however, faces the issue of leaking memory allocated during the execution and also leaving locks unreleased which could potentially lead to a deadlock in the kernel. Keeping track of the resources acquired during a BPF execution. This can be achieved by saving metadata for each allocation and de-allocation which can be referenced during termination to cleanup all resources which are found to have no complementing de-allocation entry. Since BPF is used in several latency-intensive applications, incurring a runtime overhead for tracking resources, specially for no-termination cases, is not desirable.

With above requirements and design goals, we propose the Fast-Path termination mechanism in chapter §4 where we leverage the resource management embedded within every BPF program due to the verifier's guarantee of resource cleanups(ref. § 2.1), to provide termination

with zero-overhead for no-termination case.

### 2.2.3 Tying it all together

We describe how the above two solutions will help a BPF Orchestrator or any client to enforce runtime policies for BPF programs in a system.

In the new scenario, an operator would set some performance policies to protect SLAs for critical code paths in their system. A BPF Orchestrator, responsible to enforce these policies, will provide admission control where it will deny installing BPF programs which the Runtime Estimator could deterministically prove to exceed the set performance policies. For cases where the the BPF program depends on dynamic parameters, the orchestrator will allow installing the program. However, during monitoring, if any of the installed BPF program is found to be the reason for exceeding the latency threshold for a given critical path, the orchestrator will issue a termination and remove the BPF program to quickly restore performance of the critical path. The policies can be made smarter to differentiate between a core-functionality BPF program, say performing system call filtering, and a non-critical tracing and monitoring BPF program, such that in the event of a performance policy violation, the BPF program with least priority in the critical call-path will be uninstalled (in this example, the tracing BPF programs).

## 2.3 Conclusion

In this chapter we described the BPF subsystem and proved how the BPF's termination guarantee is not enough for production systems. We explained using few example BPF programs which, though being verified, showed a runtime of several seconds upto several hours.

This motivated the need of a preventive solution where an operator can detect long-running BPF programs ahead of installing them, and prevent them from affecting performance of the system. For BPF programs which cannot be determined to have high-latency, at load-time, we motivated the need of an efficient termination mechanism which can abort a BPF program without waiting for the program to exit itself. Together, the runtime estimator and the termination mechanism, will facilitate an operator or a BPF Orchestrator to enforce certain class of policies, mainly performance policies, to protect from high-overhead BPF programs from disrupting SLAs for critical-functionalities of a system. In the next chapter, we describe our design for the Runtime Estimator.

# Chapter 3

# BPF Runtime Estimator

In this chapter, we discuss the Runtime Estimator which is our proposed approach to prevent long-running BPF programs from being installed into a critical path into the system. We first understand the underlying challenges involved in designing such a performance predictor for BPF programs.

## 3.1 Challenges

While the restricted complexity of eBPF [1] makes static analysis feasible for BPF programs, estimating the runtime of BPF programs still poses challenges that are unique to BPF:

1. **Multiple program paths**: BPF programs can have complex branches distributed across several object files[52] which can be missed during dynamic profiling.

2. **BPF programs do not convey the complete picture**: BPF programs depend on helper functions that are opaque to the verifier during verification, i.e. the verifier cannot step into them for performing analysis with given parameters. As BPF programs

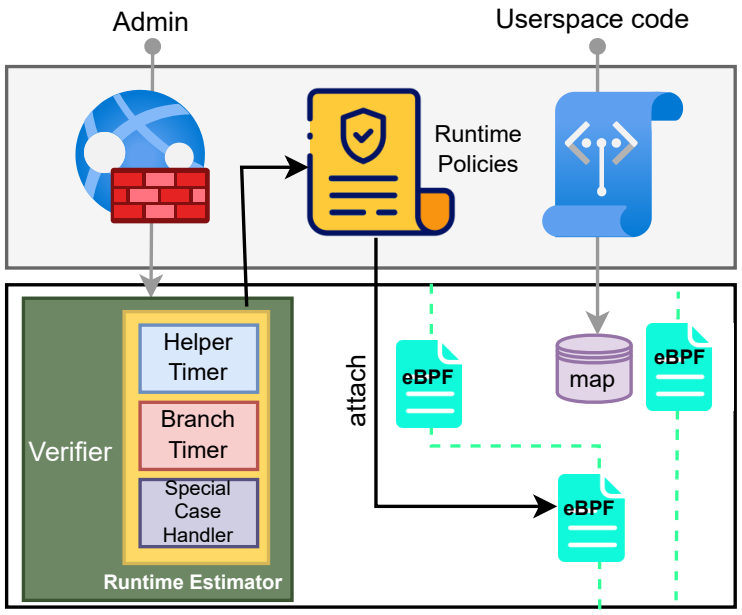


Figure 3.1: The BPF-orchestrator uses worse-case estimates from the enhanced verifier to regulate latency across critical paths

frequently use these helpers, which internally can be performing complex operations, their contribution to a program's latency cannot be ignored.

3. **Control flow changes due to helpers**: Helper functions like *bpf_loop* can dictate how long a program will run based on the number of iterations. With more helpers like the BPF inlined-iterators being introduced [42], the proposed solution will need to consider the influence of these helpers on the runtime estimation.

## 3.2 Estimating BPF Runtime

Our key insight is that the eBPF verifier, already, traverses all possible branches of a BPF bytecode to perform range analysis over the registers being used to ensure that none of the possible branches breach safety properties. Our proposed solution leverages the existing

infrastructure of the eBPF verifier and uses the verifier's Control Flow Graph(CFG) analysis to iterate through all possible branches. As the verifier is not capable of iterating into helper calls, we propose using a hybrid approach using runtime measurements of the helpers in tandem with the CFG generated by the verifier.

In the following sections, we first elaborate on the design of the proposed Runtime Estimator. From the design, we find some helper functions to show a very wide range of runtime, which we further explore in section §3.2.2.

### 3.2.1 The Runtime Estimator

We use a sample BPF orchestrator to demonstrate the architecture of the proposed solution(Fig 3.1). Originally, the orchestrator provided access control for installing BPF programs using, say, a simple access list. Figure 3.1 shows the modified architecture where the verifier now includes a Runtime Estimator. Based on the runtime estimates from the verifier, the orchestrator can have another set of rules, Runtime Policies, which determines whether the potential overhead from the BPF program would degrade the system performance beyond a permissible threshold.

The Runtime Estimator internally has three sub-components: the helper-timer, the branch-timer, and the special-case handler.

**The helper-timer**: This component is responsible for creating a mapping between all available helper functions and their respective best and worst runtimes, using offline measurement, at boot-time. We assume that helper function runtimes are deterministic and well-defined. To obtain the estimates, we use BPF samples from the Linux kernel repository with the required helper and attach them to the intended hook. The samples were modified such that each helper was invoked 1000 times to report the best, average, and

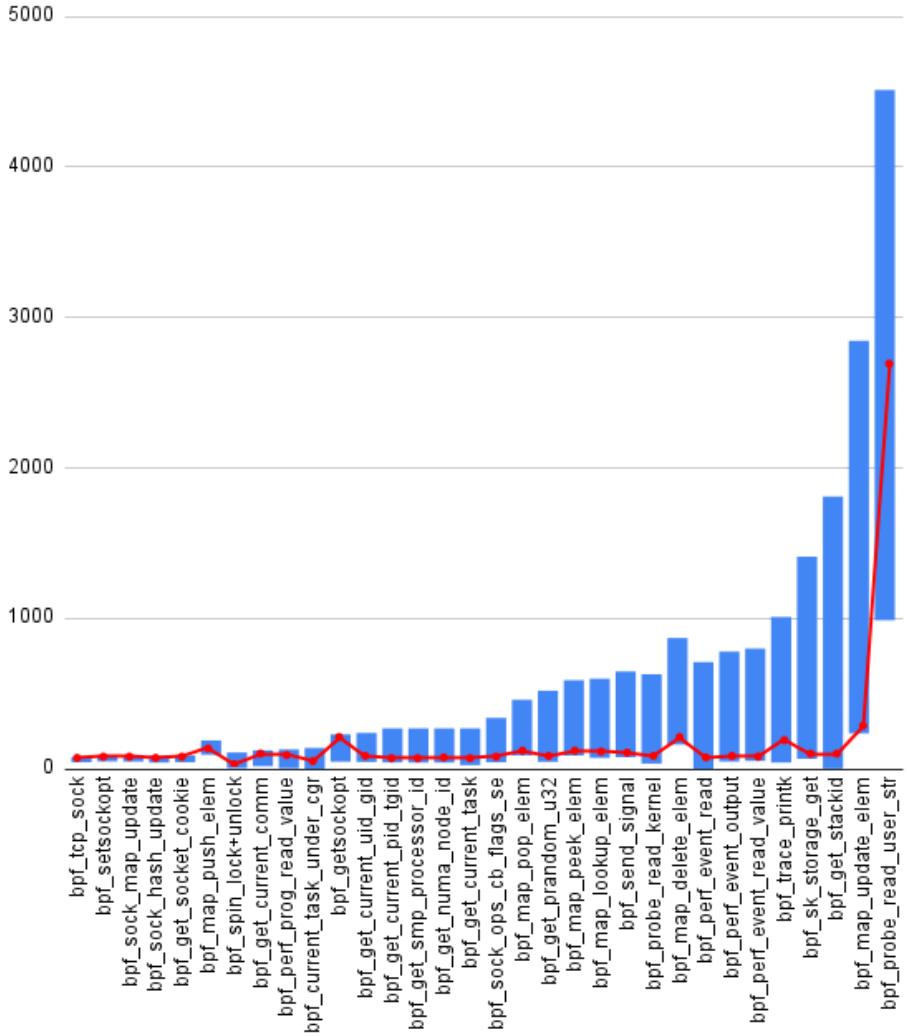


Figure 3.2: Runtime range of BPF helpers (ns). Each bar represents best-case to worst-case runtime. The line graph shows the average runtime for each helper.

worst-case execution times. The modified sample BPF programs were triggered 10 different times to capture performance across varying system loads. For map-based helpers, we used the LRU hashmap (BPF_MAP_TYPE_LRU_HASH). For measuring execution time, we used the *bpf_ktime_get_ns* helper and assumed its variance to be negligible.

Figure 3.2 shows the runtime estimates obtained for 31 helper functions. The length of the bars describes how some helpers like *bpf_tcp_sock* show a tight bound on runtimes while helpers like *bpf_map_update_elem* and *bpf_probe_read_user_str* show a high variation due to dependency on an argument or because of lock contention, which we further discuss in section §3.2.2. The average runtime (denoted through the line chart) is usually close to the best-case runtime even if the best-to-worse case gap is high.

During this evaluation, we occasionally observed very high worse-case runtimes reaching about 100-400x the otherwise observed worst case. Based on the rarity of these data points and their relationship with how long a helper executes, we speculate that bursts of interrupts from events like network packets or timers were the reason. As these factors are not specific to BPF, we omit these outliers from our helper timings and Figure 3.2.

**The branch-timer**: This component uses the verifier's iteration over the BPF bytecode to detect all possible branches. For a given branch, all helper calls are referenced with the mapping generated by the helper-timer to update the state variable containing best and worst runtime estimates. If the helper is observed to vary in a well-defined relation with its arguments, the branch-timer can adjust the estimates.

**The Special-case handler**: This component is responsible for adjusting the theoretical runtime estimate to match control flow changes by helpers like *bpf_loop*. For the *bpf_loop* helper, which takes in the number of iterations and the static function to iterate over as arguments, the special-case handler determines the iteration count stored in register *r1* and

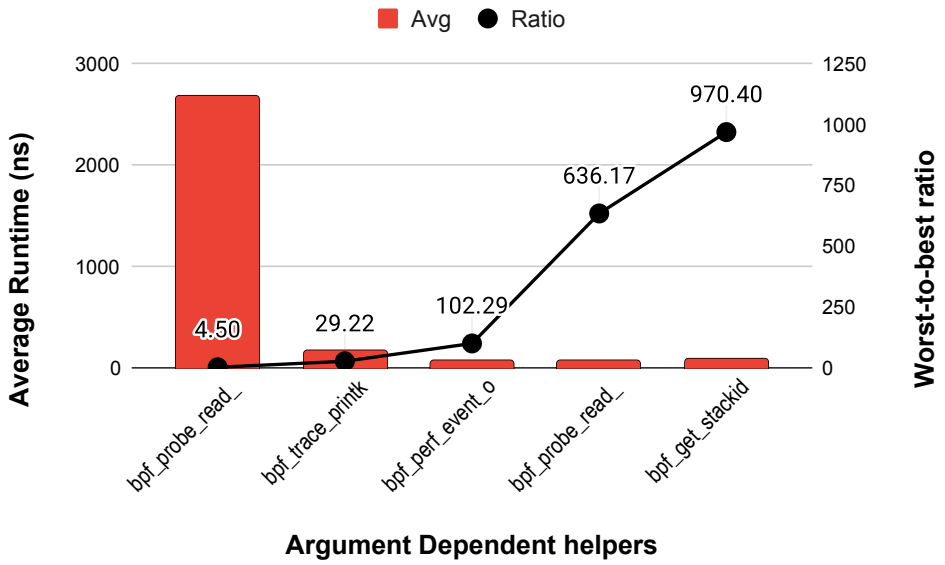


Figure 3.3: Some argument dependent helpers show a drastic increase in runtime (900x) with varying argument values

uses it with the estimated runtime of the static function to suitably increment the estimate of the overall program. This special case handling can be suitably extended to other loop-based iterators like *bpf_for_each* and *bpf_iter*, which can non-trivially influence the control flow of a BPF program.

At the end of verification, the Runtime Estimator reports the overall minimum and maximum runtime value over different branches as the global runtime estimate. The BPF-orchestrator is informed of the obtained runtime estimates which can now be checked against additional runtime policies before attaching the eBPF bytecode.

### 3.2.2 Categories of long running helper

From our previous experiments (Fig 3.2), we observed certain helper functions to have a broad best-to-worst case runtime. In this section we categorize the BPF helper functions

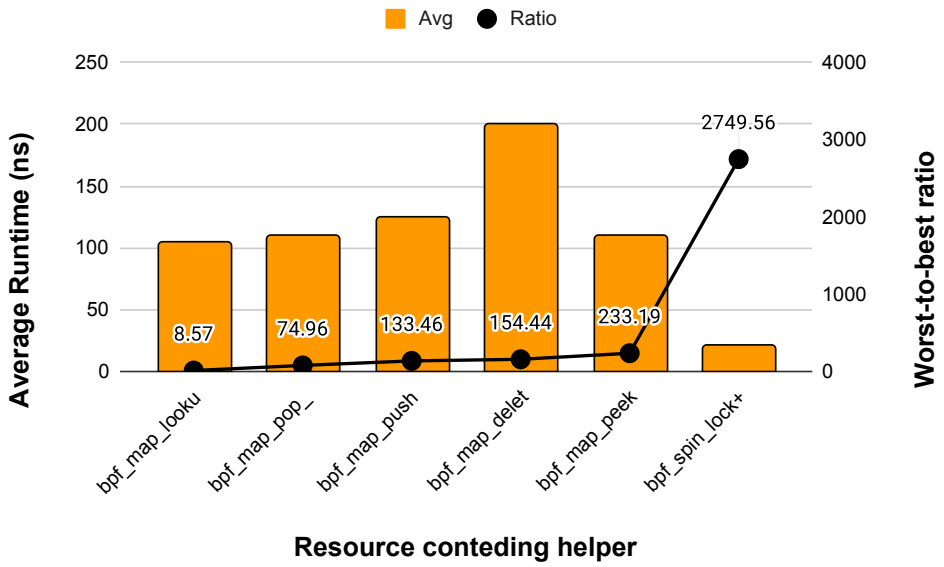


Figure 3.4: Resource contending helpers show a higher volatility where the worst-case runtime showed an increase of 2500x when several CPUs contended for a shared map access.

based on the root case of their tendency to exhibit long runtimes.

- **Argument-dependent helpers**: Certain helper functions like *bpf_perf_event_output*, *bpf_get_stackid* and *bpf_probe_read_str* exhibit a runtime which is proportional to the arguments passed during runtime. For example, the *bpf_perf_event_output* helper, which copies a given blob of data to a BPF map, will need to perform iterations proportional to the size of the given data. Upon performing basic tests with a range of possible arguments, we observed the worse-case runtime to be about two orders of magnitude higher than the best case(Fig 3.3). We explain more about the test and the test environment towards the end of this section. If the arguments are determined dynamically at runtime, the call can show an unexpectedly high execution time.

- **Resource-contending helpers**: BPF programs that need to share information with the userspace or with other BPF programs use map helper functions to read/write values. For example, the *bpf_map_update_elem* helper is used to insert/update a key-value

pair in a BPF map. Depending on the map type and also whether it is globally shared or unique per-CPU, the map helper acquires a series of locks to provide concurrent-safe updates[5]. Helpers like *bpf_spin_lock* are more direct in terms of resource contention where the helper provides the spin lock/unlock feature that can contend if some other BPF program wants to acquire the same spin lock. Both the examples above show 2-3 orders of magnitude increase in runtime at worst-case (Fig 3.4).

- **Iterators**: To support functionality like bounded loops and data structure iterators, helpers like *bpf_loop* and *bpf_find_vma* exist. The execution time of these helpers depends on the iteration count and/or size of the list being traversed. For example, the *bpf_loop* helper provides a for-loop functionality to execute a piece of code for several iterations. Given the high value of the maximum allowed iteration count, a dependent BPF program can experience delays due to an unexpectedly high iteration decided dynamically. Another example is the *bpf_find_vma* helper which performs a VMA lookup for the object passed as an argument. If a workload is allocating several small chunks of memory ending up with an unusually large VMA list, this helper call will exhibit an unexpectedly high execution time than expected.

  Other iterative helpers function like *bpf_for_each_map_elem* and *bpf_user_ringbuf_drain* issue a callback for each element iterated from the list, and thus take longer to execute if the callback function has resource intensive operations such as atomic updates, other expensive helper calls, etc.

To quantify the gap between the best case and the worst case runtime of above discussed helper functions, we created a test bench with Linux kernel 6.0.2 in a QEMU environment with 2 CPUs and 2GBs of RAM on a 12th Gen Intel(R) Core(TM) i7-12700 processor. For argument-dependent helpers, we executed the helper functions with a wide range of possible inputs to determine its best-case and worst-case runtime. As shown in Fig 3.3, these helper

functions can show a worst-case runtime of several hundred times the best-case runtime. For evaluating the performance of resource contending helpers(Fig 3.4), we loaded a BPF program to a tracepoint in two different CPUs by loading the program twice and used the *taskset* command to pin them to a CPU each. In the case of the map lookup and update helper, the BPF program contains a simple lookup/update on a shared BPF map for a pre-defined key. Upon each tracepoint event, both the BPF programs will attempt to read the pre-defined map element concurrently. To maximize the concurrency overhead, we chose a global map of type LRU hash where the global scope and the LRU ordering imposes more overhead for each concurrent access. The obtained results for worst-to-best runtime ratio as shown in Fig 3.4, is more extreme than that for the argument-dependent helpers and goes to be about 3 orders of magnitude in the worst case for the spin lock helpers. In terms of absolute figures, the highest runtime these helpers could reach was about 50,000 ns for the *bpf_spin_lock* helper function.

### 3.2.3 Limitations

We now discuss the limitations of the proposed Runtime Estimator.

- **Helper Runtime measurements are incomplete:** Dynamic measurements of helper functions face the issue of being incomplete. The experimental setup consisted of two variation:
  - Varied inputs to simulate different function arguments.
  - Varied amount of parallel executions to simulate congestion for cases like BPF map and spinlock helpers which contend on shared locks.

  While the above two type of input variation gave a good view about the runtime trends of the different helper functions, the numbers do not account for all possible cases. The

resulting BPF program's estimates, being derived from underlying helper's estimates, will not be complete and thus could be an under or an overestimate of the actual runtime.

- **High variance of helper estimates:** High variance in runtime performance of helper functions (Fig 3.3, Fig 3.4) implies a broad performance estimate even if the actual runtime is mostly showing the best-case runtime. For example, a simple XDP program which performs a map access to check for packet-drop rules will have a performance estimate with a much higher worst-case value than the actual runtime value the program exhibits majority of the time. Thus, while having a wide-but-complete range allows better runtime policies on one hand, it is not very useful for an operator who observes the high-runtime to be rarely ocurring and thus ends up installing that program anyways.

## 3.3 Summary

In this chapter, we proposed a BPF Runtime Estimator which used static analysis of the verifier with offline measurements of helper function to statically predict the range of runtime which a BPF program would show when executed. Using the proposed estimator, a client like a BPF Orchestrator can enforce latency-based policies where a BPF program exceeding the allowed latency for a given hook point can be rejected to keep the total overhead on hot-paths to be under a strict limit. We found certain classes of helper functions which show runtime based on dynamic factors which are not known statically at program load time. This is a limitation of the proposed Runtime Estimator due to its static nature. Thus, in the next chapter, we discuss the BPF termination solution which provides a runtime enforcement mechanism for BPF programs which cannot be detected during program installation to have

long runtime.

# Chapter 4

# BPF Termination

## 4.1 Termination Background

To terminate any program, there are two broad issues:

1. Are all resources acquired during execution released?

2. Is the system left in an unexpected state that could end up affecting other programs or future invocations of the same program?

For userspace programs running over an OS, an administrator does not have to worry about issue #1 because the kernel reclaims resources at the end of a process. However, for applications like a server that is expected to run for years without reboot[54] and with several components, resource leaks due to multiple instances of termination can eventually culminate in an out-of-memory issue, which can cause the server to crash. Regarding issue #2, if termination leads to some internal states left at an intermediate value, then other parts of the system can end up in a faulty state. In case of an unreleased lock, another program

using that lock will then starve because of a deadlock. Hence, to support safe termination, both the above issues need to be addressed.

Most modern programming languages have resource cleanup as a critical component to address issue #1. In Rust, the language supports two primitives to prevent resource leaks: *Resource Acquisition Is Initialization*(RAII) and *landing pads*. RAII is a paradigm that is used by the compiler to ensure all objects allocated in a program are destructed once no more needed. Landing pads, on the other hand, directly address the issue of cleanup in the case of an exception by providing all required destructor calls to which the execution can jump and release the leftover objects. Since some resource allocation could be branch-dependent, the landing pads also contain branch-based de-allocation for which the language tracks all branching decisions taken at runtime. The cleanup mechanism is similar for C++ storage classes like shared and smart pointers, where the GCC compiler generates a separate code path in the ELF executable(namely the *gxx personality* routine) that serves as the unwind code.

### 4.1.1 Resource cleanup in BPF

BPF programs, however, are typically written in the C programming language which is translated to a simplified BPF bytecode and is eventually converted into the native machine code, such as x86. Since the C language has no resource management, it is left up to the programmer to ensure resource release and system safety.

While the industry-accepted way of generating exception handler routines(like Rust's *landing pads* and C++'s *gxx personality*) can be ported to the BPF subsystem and possibly work correctly, it will cause a giant code footprint not just in the Linux kernel code[22], but also in other compiler toolchains involved in the code generation process. In the next sections, we talk about the intricacies unique to BPF which will dictate the design decisions in our

final proposed approach for supporting termination.

## 4.2 Termination for BPF

To safely terminate a program, we require the list of active resources to release(section 4.1). BPF being written in C, lacks this required piece but looking closely BPF programs have more information in terms of resource allocations compared to a simple C-based kernel code. Since BPF programs need to go through the verifier's static analysis, BPF becomes a "managed" language that supplements the original language with properties that were otherwise not available. In the case of BPF, these properties enforce resource management by the verifier's bookkeeping of resources. While this bookkeeping was originally meant to prevent resource leaks, it can be used to track all active resources at any random point of a program execution which could then be cleaned up if termination is required. The challenge then evolves from *"What resources to clean up?"* to *"How to efficiently use the verifier's knowledge to perform cleanups?"*

However, BPF programs tend to use kernel-provided interfaces called helper functions which do not have a similar list of active resources because the verifier treats helper functions as black boxes. Thus, issuing a termination request for a BPF program without being mindful of whether the CPU is currently within the BPF context or the kernel context (through helper function) can put the system into an inconsistent state for reasons like an unfreed lock. Therefore, to guarantee safe termination, the runtime needs to be aware of the context i.e. whether currently in BPF context or kernel context. This brings us to the second challenge which is to correctly identify the current context and defer termination until execution reaches back to the BPF text.

## 4.3 Fast-Path Termination

An early termination mechanism needs to satisfy the following requirements:

- Support uninstalling a BPF program pre-maturely i.e. the uninstall should not have to wait till an ongoing program execution completes.
- Termination should not break the safety properties of the original verified BPF program.
- A termination request should not interfere with core kernel execution to not leave kernel objects in an inconsistent state.
- Avoid the complexity and overhead of creating landing pads.
- Minimize the time and space overhead due to a termination mechanism.
- Provide an option to terminate all instances or a particular execution instance of a BPF program.

We introduce Fast-Path Termination which leverages both the memory safety and termination property of eBPF to provide a simple termination mechanism for BPF programs. The key insight of Fast-Path termination is that BPF programs will always have implicit resource cleanups encoded through one or more program paths. As the verifier performs static analysis to ensure all possible branches release all resources before exiting, rewriting the BPF binary to remove costly future helper calls and resource allocations can lead to a great reduction in the execution time of the program effectively terminating the program early. This process of patching the BPF program will still carry the same safety properties as promised by the verifier over the original program. The proposed design is broken into a preparation stage and an enforcement stage. Figure 4.1 illustrates the preparation stage

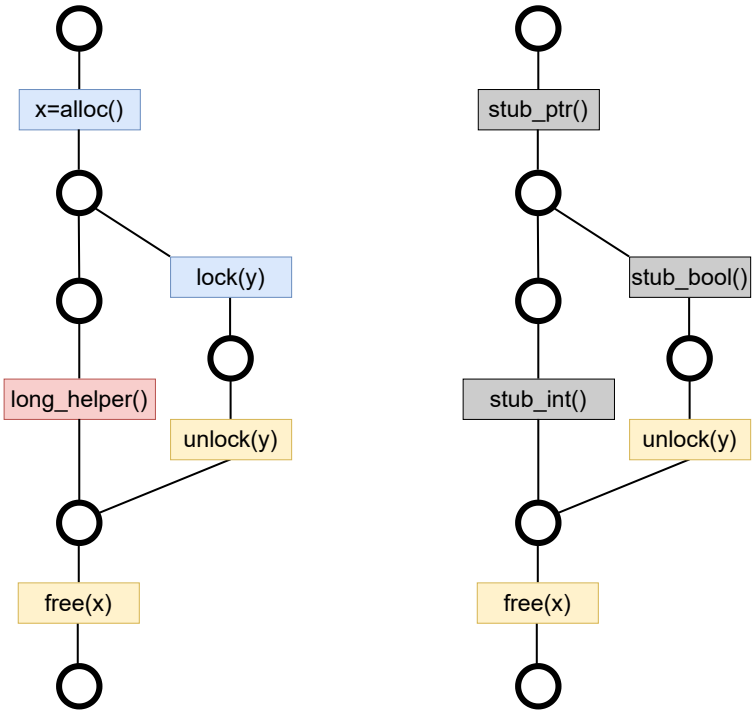


Figure 4.1: Any helper that is not releasing resources will be stubbed out in the termination copy of a BPF program

which involves processing a new program load request and creating a termination copy for it:

1. Create stubs in kernel code that will be used to replace costly helper functions. To satisfy the verifier, we need a new stub for every possible return type of helper function. For example, stub_int() replaces integer return type helper, stub_ptr() replaces pointer type helpers, etc. These stubs will be empty functions that will quickly execute and save execution time when replacing the original costly helper calls.

2. Once a user tries to load a new BPF program that invokes the verifier, perform the two modify operations:

   (a) Go through all helper and kfunc calls and replace them with the relevant stub function. While doing so, make exceptions for functions releasing resources. By

the end of this process, we obtain a copy of the original BPF program but a very lightweight version of it due to no helper functions remaining.

(b) To tackle long-running iterative helper functions like bpf_loop and bpf_for_each_map_elem mentioned in section 2.2.2, our observation is that these iterative functions already support return value based execution control, Only when the previous callback execution succeeds, the iterator continue to the next element. We can leverage this existing feature to tweak instances of such iterators present in a BPF program. In this step of program modification, we simply modify the return value of the associated callback function to always return a failure. This will make the iterative helpers quickly exit the loop as soon as they invoke the modified callback function.

3. Pass this new program through the verifier to prove safety.

4. Any BPF program installation will now lead to two programs being added to the kernel text: the original and the termination copy.

### 4.3.1 Patch Generator

A BPF program during termination, will see $n$ logically different programs if there are $n$ helper calls in it. The first being when the program just started to execute but got terminated before it could execute a single helper. In Figure 4.2, this is represented by Patch #n. Subsequently, if the program could execute just one helper before termination, it will logically be running the program shown as Patch #n-1. Extending this, the final case will be when the last helper call is stubbed due to the termination signal as represented by Patch #1. To ensure that the safety properties of all logically possible programs are upheld, we modify the program loading step to now call for verification of all possible programs along

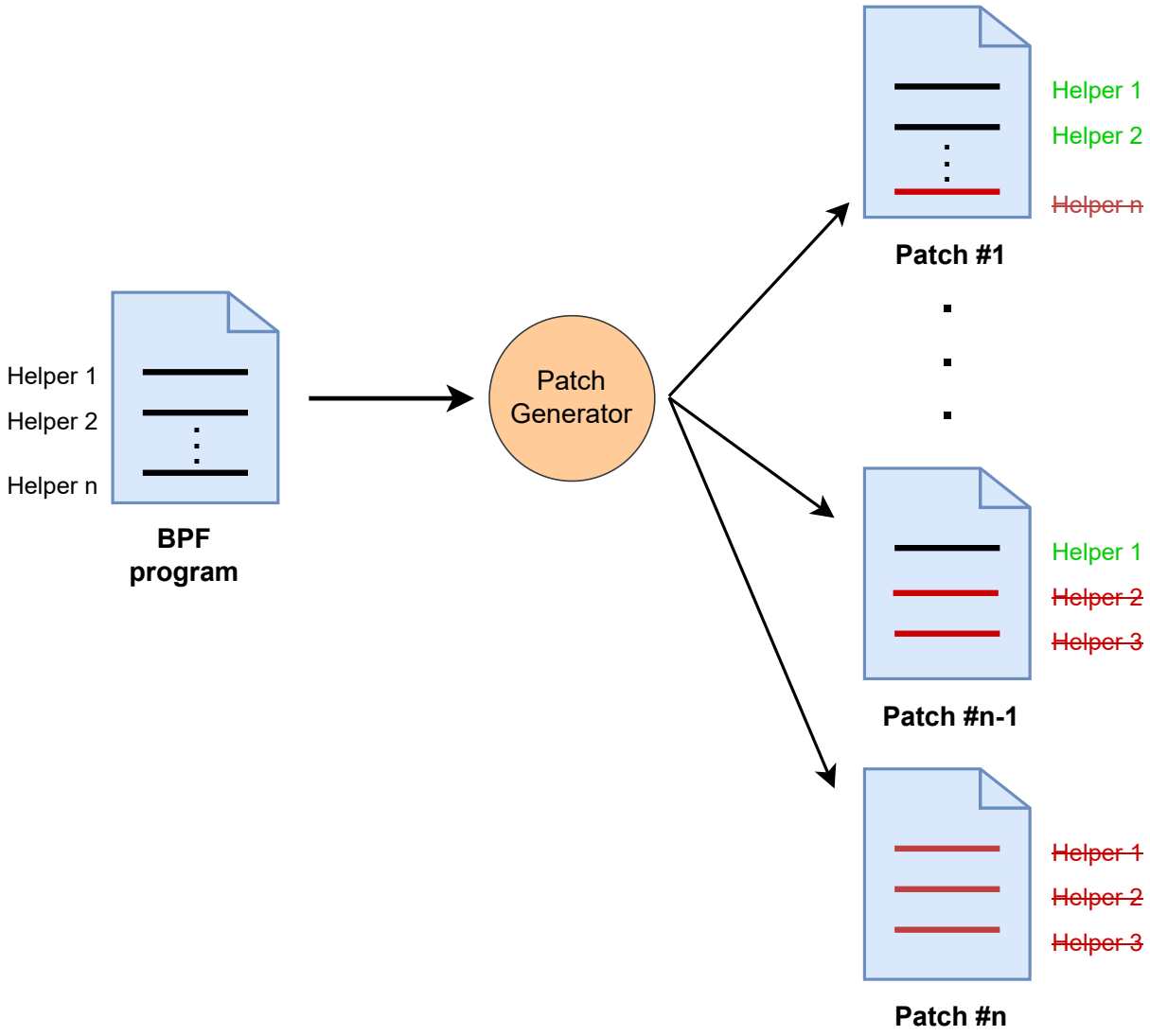


Figure 4.2: The final patch i.e. Patch #n, if verified, is used as the termination copy

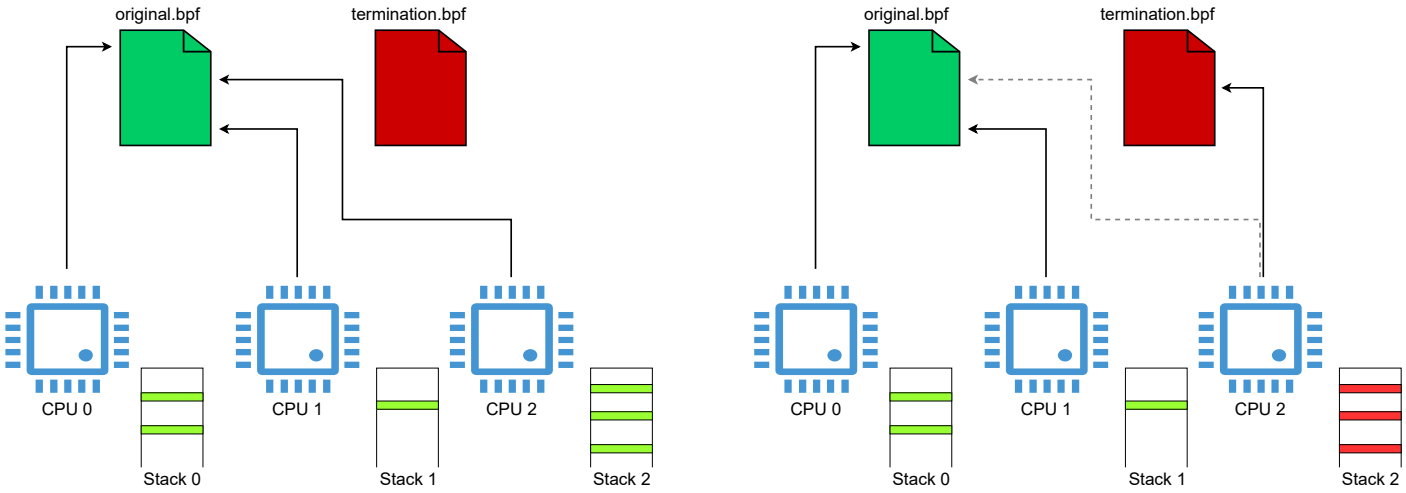


Figure 4.3: Every BPF installation will create a termination copy prepared.CPU 2 migrated to the termination copy. The corresponding stack is also modified to suitably change all the return addresses.

with the verification of the original BPF program with the help of Patch Generator. The Patch Generator is responsible for finding all helper and kfunc calls and replacing them with the corresponding stub as explained before. The generator will also carefully skip stubbing any resource-releasing calls as per the discussion above.

### 4.3.2 The complete flow

With the above setup ready, any termination request will have to follow the steps in the enforcement stage:

1. Unlink the BPF program from its attached point to stop invocations succeeding a termination request.

2. If termination is requested with system-wide effect, overwrite the original BPF text with the termination copy. Ensuring atomic writes is necessary to prevent execution of half-modified instructions as all CPUs will be executing without any interrupts.

3. If termination is requested for a particular instance, the procedure follows two steps as shown in Figure 4.3:

   - Interrupt the target CPU to move its Instruction Pointer register to point to the relevant offset in the termination copy already available in the kernel.

   - Since the new termination copy will have a different memory address, perform a stack walk on the target CPU's kernel stack and modify all available return values that are pointing to the old BPF memory addresses.

     The stack modification logic uses the ORC unwinding in the Linux kernel, which is configured to use the ORC debugging headers generated during compile time. The unwinder is called to obtain the return address of the current call stack until the base of the stack is reached. For each return address found to be within the text region of the target BPF program, the new offset is calculated and the corresponding stack value is replaced.

   At the end of these two steps, the target CPU will start executing over the termination copy of the BPF program and quickly terminate. However, the rest of the executions across the system and also future invocation in the current CPU will keep running the originally installed BPF program.

With the above approach, the proposed fast-path mechanism will quickly skip over the costly functions of a BPF program and when combined with the knowledge that BPF programs have a limited program size imposed by the verifier, the time to execute the rest of the modified program will be very low.

### 4.3.3 Upholding safety properties of termination copy

The termination program copy cannot break any safety properties of the BPF verifier due to the following reasons:

- While short-circuiting helpers leads to returning error values, such as NULL for pointer return type functions, forcing such an error response cannot break any safety properties as the verifier always assumes a helper's return value to be all possible values for the data type until the programmer explicitly checks for errors/boundary conditions. Thus, if a map access helper call gets stubbed out and always returns NULL, the verifier's check to ensure a programmer is not dereferencing NULL will handle the NULL returned by our helper stub. Similarly, replacing an integer return type helper with a stub returning, say, -ETERM, cannot break any safety properties.

- Changing the return values of the callback functions (mentioned as the second type of modification required) will only make the program take a different program path if the program has any check for an early break from the iterations. Since the verifier would have already traversed all possible program paths, our modification will only make the program flow through one of the anticipated program paths, which would always be safe!

### 4.3.4 Explicit helper return codes

There is a case for a safety violation that demands attention. By stubbing all resource-allocating helpers, we are left with all corresponding resource-releasing helpers which could be expecting a valid and non-NULL object to be passed. This could lead to a NULL pointer dereference when the release helper call tries to read the fields saved in the passed ob-

ject(which is NULL) and try to free them. Currently, the design pattern mandates an exit the moment an allocation fails, i.e. returns NULL for most helpers. However, the pattern is not followed for the case of bpf_spin_lock helper functions where there is no error check at the time of locking, due to which the unlock helper also assumes the passed lock object will always be valid. We believe this example to be an outlier when compared with the design pattern for all other helper functions which can be easily made to conform by adding a few lines of code.

# Chapter 5

# Evaluation

## 5.1 Setup

All the experiments were performed on a 12th Gen Intel(R) Core(TM) i7-12700 machine with 20 logical CPUs and 32GB RAM on Linux kernel version 5.15 and Ubuntu 20.04.

## 5.2 Evaluating performance estimates

In this section, we use the observations made during helper-runtime calculations(§3.2) to evaluate the correctness of the Runtime Estimator as a whole. We ran 13 different eBPF programs from the `samples/bpf` directory of the Linux kernel through the modified verifier which gave out a probable runtime range for each BPF program. For actual runtimes, the runtime statistics saved in the kernel were referred to. All the experiments were performed on a 12th Gen Intel(R) Core(TM) i7-12700 machine with 20 logical CPUs and 32GB RAM on Linux kernel version 5.15 and Ubuntu 20.04. Obtained values are listed in Table 5.1.

From the table, we can infer that the actual runtimes always lie within the expected runtime range produced by the modified verifier. While the actual runtimes are within a 50% of the worse case for the *tracex1*, *sockex1*, *trace_event* and both the *tcp_basertt* and *tcp_dumpstats* samples, we see a big margin for rest of the samples. When we looked through the code of these samples, we observed them to predominantly use high-variation helpers including *bpf_map_update_elem*, *bpf_probe_read_kernel* and the *bpf_get_stackid* (ref Figure 3.2). Note that the *bpf_map_update_elem* estimates were based on the LRU hashmap for the purpose of observing worst case execution while in the actual evaluation samples, the *BPF_MAP_TYPE_ARRAY* and *BPF_MAP_TYPE_HASH* map types were used. We expect our estimates to significantly improve when re-run with the actual map types. The other two helpers mentioned before depend on parameters which are mostly unknown at verification time. To tackle this problem, we can assume a median parameter value to get the most probable runtime which a parameterized helper can take.

We also performed the experiments using average-case helper runtimes (figure 3.2) and observed estimates much closer to actual runtimes. This is owed to fact that most of the helpers showcase an average runtime which is much closer its best case runtime despite showing a high tail latency. While our framework supports average-case estimates, we find that the worse-case values gives a better picture as they guarantee an upper-bound latency needed for policy-based decision making.

To evaluate the special-case handler and the overall Runtime Estimator, we wrote a BPF program (ref. Listing 5.1) with nested *bpf_loop* helper calls. We use this simple program to demonstrate a rare branch which can be easily missed in dynamic benchmarking. In the pseudocode, The *bpf_get_random_u32* helper generates a 32-bit random number based on which either of the two branches will be followed. A dynamic benchmarking will highly likely pursue the *if-condition* as the probability of getting ($key < 10$) is very low for a 32-bit

| Sample Program | Expected Runtime | Actual Runtime |
|---|---|---|
| tracex1 | 192-2660 | 1011 |
| tracex2 | 48-4028 | 88 |
| tracex3 | 249-2040 | 468 |
| tracex4 | 48-3454 | 279 |
| tracex5 | 48-800 | 123 |
| sockex1 | 86-590 | 225 |
| sockex2 | 86-3424 | 551 |
| sockex3 | 86-4274 | 123 |
| test_current_task_under_cgroup | 315-3224 | 753 |
| test_probe_write_user | 48-2450 | 761 |
| trace_event | 171-7878 | 6252 |
| tcp_basertt | 171-2220 | 1039 |
| tcp_dumpstats | 57-3470 | 2277 |

Table 5.1: Expected and actual runtime of eBPF samples

```
1 int simple(void):
2     bpf_printk("A rare number")
3
4 int loop_simple(void):
5     bpf_loop(100,simple)
6
7 int main():
8     key = bpf_get_prandom_u32()
9     if (key>10)
10        bpf_printk("A common number")
11    else
12        bpf_loop(10000,loop_simple)
```

Listing 5.1: Pseudo-code showing nested loops with branching in a BPF program.

randomly generated integer. Using the Runtime Estimator, the verifier generated a range of 115 to 180,000,510 nanoseconds where the worst case belongs to the branch containing nested loops. When this program was attached and triggered, we observed an actual runtime of 125,013,362 ns when the worse case branch was pursued. Hence, using the proposed solution, the verifier identifies the most expensive branch and correctly reports it.

## 5.3 Evaluating BPF termination

In this section, we check the working and then evaluate the correctness and efficiency of the proposed runtime termination mechanism. To make use of the recently added helpers and changes in verifier limits, we use Linux kernel version 6.0.2 for the evaluation.

We started with the aim to design a termination mechanism to kill long-running BPF programs which before this work, had no way to be stopped from throttling system performance until the program finishes (section §2.2.2 Fig 2.3). Using the proposed termination approach in §4 we rerun the experiment (figure 5.1) to show how the proposed termination mechanism enables a BPF program to return significantly earlier than its expected completion time. The

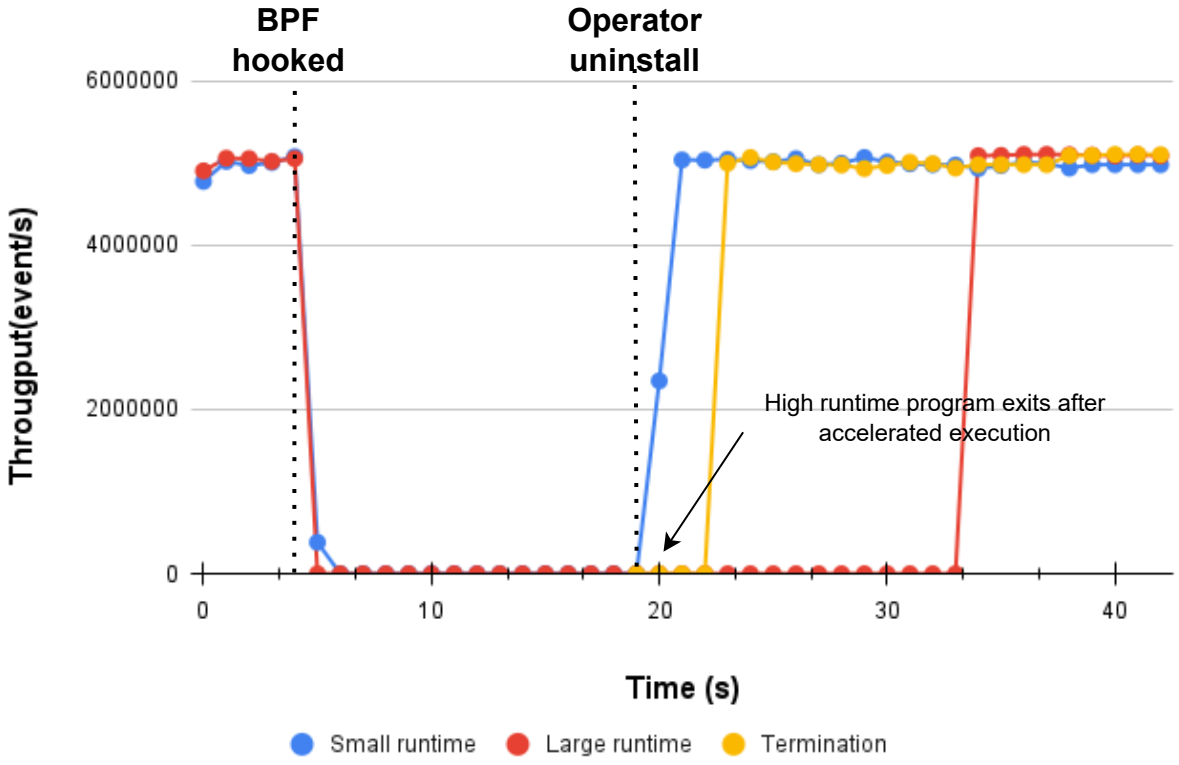


Figure 5.1: Upon receiving the termination signal, the runtime waits for the last running helper function to finish before removing the BPF program.

observed delay in restoration of performance of the termination line (green) is due to the

| Case | Throughput (calls/s) |
|---|---|
| Syscall | 5106664 ± 4.7% |
| Syscall + BPF | 4258541 ± 4.1% |
| Syscall + BPF + Termination | 4302707 ± 4.0% |

Table 5.2: Introducing termination brings practically no difference in throughput.

time taken to let an already executing helper function return to the BPF program. This is in accordance with our design principle where we do not want to terminate a helper function, which is a kernel code, as it could otherwise lead to unreleased memory and locks.

### 5.3.1 Overheads of Termination

After demonstrating the working of the proposed termination mechanism, we will now evaluate the different overheads introduced into the Linux kernel to support BPF Termination. We break the overhead into the following categories:

- **Runtime overhead**

  To provide termination, every BPF invocation needs to log its current CPU ID which is later used by the termination call to identify the target CPU. To measure any overheads due to the termination mechanism, we perform the following benchmarking:

  1. We create an empty system call and repeatedly call it to find the average number of calls per second and use it as our throughput value.
  2. Create an empty BPF program and attach it as a tracepoint to the system call created in the above step.
  3. Recompile the kernel after enabling BPF termination and perform the throughput measurements again.

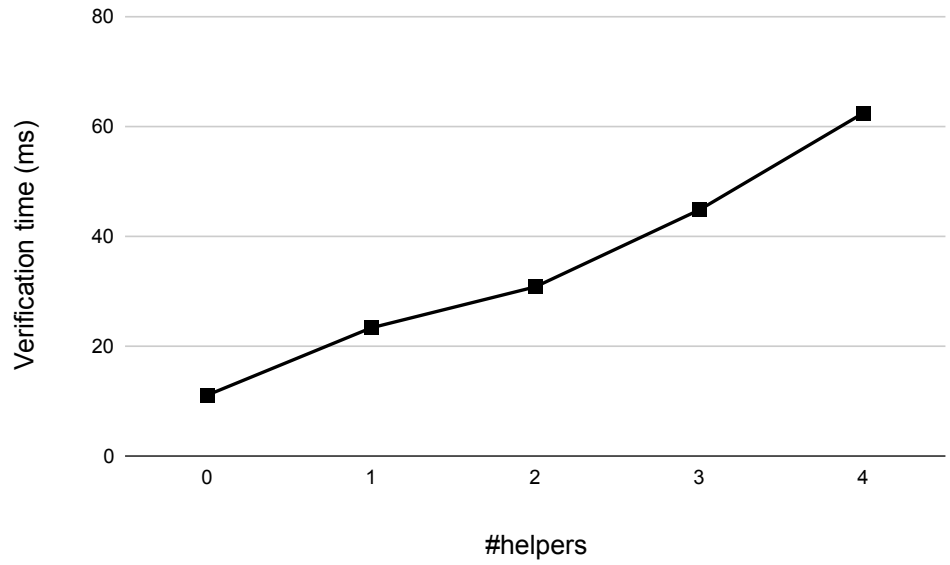


Figure 5.2: Verification time varies linearly with the number of helper calls in the BPF program

The measured values are shown in Table 5.2. We find the overhead to be negligible because of the minimal number of instructions added to perform the required operation to store CPU ID. The design can be modified to omit this overhead by sending IPI to all available CPUs and letting each interrupt handler decide if their host CPU is currently running the target BPF program. While such a design would save the constant runtime overhead we observed in table 5.2, sending IPIs to every CPU in a system might not be desirable, especially on a system with hundreds of CPU nodes.

- **Verification Time** The proposed termination mechanism adds verification time as it needs to verify all possible patches which the original BPF program can logically get into due to an abrupt termination request. The verification overhead due to each additional helper call in a BPF program comes from two sources:

  1. Deep copying the original BPF program's internal structure (*bpf_prog struct*) to create a clone and modify the clone to patch the helper call.

2. Passing the clone to the verifier for static analysis.

To evaluate verification overhead, we pass a BPF program with 4 helper calls through the verifier, to observe the time taken to verify the 5 different patches that will be created by the Patch Generator(section §4.3.1 Fig 4.2). As the patch generation process creates several clones in proportion to the number of helper calls, we see the verification time grow linearly as the clone has more number of helper calls to be verified(Fig 5.2). Cloning the BPF structure only copies the required fields such as attachment type, the BPF bytecode instructions passed during load time, etc. Thus, the time to clone a BPF program will mainly depend on the number of instructions in the BPF program. Due to this, the gap between each data point in Fig 5.2 will increase/decrease based on the number of instructions needed to be copied.

- **Termination Time** When an operator decides to terminate a BPF program, it issues a system call that first performs a lookup to find the relevant BPF program's data structure and issues an IPI to the CPU on which that BPF program is running. The target CPU will receive the IPI and start the termination handler. The termination handler will uninstall the BPF program from the hook point, iterate through the stack state to modify pushed return addresses to point to the termination copy of the original BPF program and change the Instruction Pointer before exiting the termination handler.

  For a BPF program to terminate, in the best case, when the Instruction Pointer was already in BPF text, a termination request will lead to just quickly executing the rest of the stubbed BPF program. The delay can be defined as:

$$termination\ delay = stub\ runtime + termination\ handler \tag{5.1}$$

where *stub runtime* is the time a stubbed out BPF program will take to reach completion, *termination handler* denotes the full loop from initiating the IPI through the syscall made by the operator, to the time taken to perform the delinking, stack modification and other steps mentioned above, until getting a response message at the end of the IPI.

However, in the worst case, the Instruction Pointer will be inside core kernel text, executing some helper function, in which case the delay will get extended by the time needed by the ongoing helper call to return to the BPF context. The delay in the worst case then becomes:

$$termination\ delay = helper\ runtime + stub\ runtime + termination\ handler \quad (5.2)$$

where *helper runtime* is the execution time possible for a helper function.

We briefly elaborate on the different sub-factors which can affect the three different components affecting the *termination_delay* which are *helper_runtime*, *stub_runtime* and *termination_handler*:

1. **Termination handler** The termination handler currently has 3 components: the unlinking code and stack walking, and Inter-Process Interrupts(IPI) (Fig 4.3). This step therefore depends on the stack depth at which the termination handler will be executed. More the depth, the more the number of unwinding operations needed. The other component i.e. IPI also depends on the runtime environment. Linux kernel currently maintains a queue of pending IPIs obtained for each CPU. Thus, if the target CPU is already executing pending IPIs, the termination signal will be delayed. To summarize, executing the termination handler depends on how many IPIs the target CPU has pending on its queue, and the depth of its

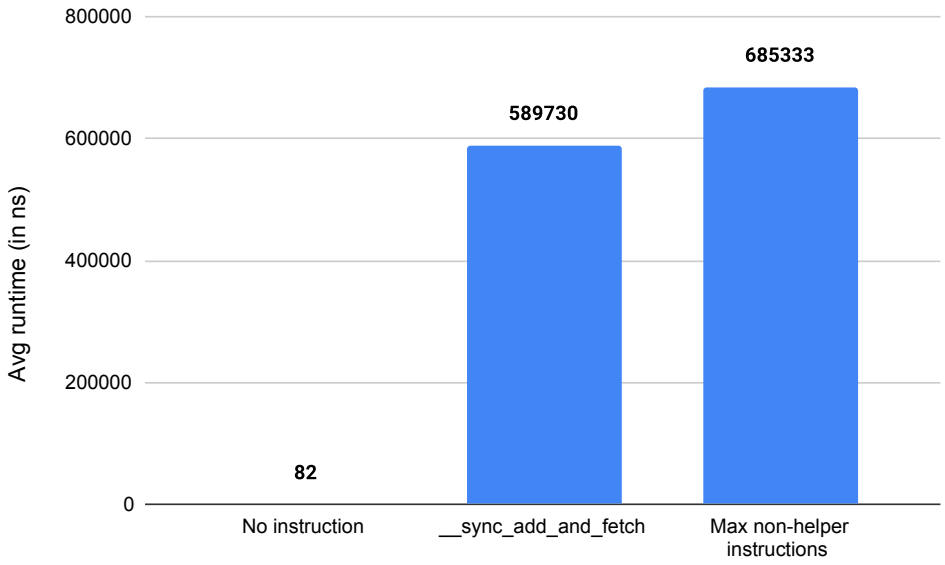


Figure 5.3: A BPF program can show a maximum runtime of ≈1ms when no helper function is used.

kernel stack.

Getting a minimum/maximum termination time is non-trivial and faces the completeness challenge due to several different factors involved:

- IPIs are handled by a per-CPU FIFO queue. If a CPU is getting a high number of IPIs at a given instant, the handlers will be waiting in the queue before being executed.
- Stack depth at the time of interrupt will determine the time required to perform the stack walk.
- Helpers with higher runtime will make the fast-path approach wait for a longer duration thereby delaying the termination.

While the above-mentioned parameters can affect the time for a termination handler to execute, we perform our experiment for an average execution environment where only the concerned BPF program is running with typical helper functions

and no external workload which would otherwise affect the IPI mechanisms as described above. To have a helper function run long enough to provide the required time window for the evaluator to trigger termination, we modified a helper function to sleep once executed. Through the evaluation, we found the termination handler to take *1255144 ns* which cannot be termed as minimum or maximum due to the issues described above. For the sake of getting a rough estimate, we use this value as both minimum and maximum in Table 5.3.

2. **Stub runtime** This depends on the length of a BPF program and the number of instructions remaining after stubbing all the helper calls present after the point where termination was requested. For example, a BPF program with mostly helper calls will be left with a much lesser number of instructions to execute after stubbing compared to a BPF program with many simple ALU instructions and relatively fewer helper calls. In the latter case, the stubbing will only reduce instructions, however, still saving the expensive helper execution. To quantify the maximum stub runtime, we perform a test where we vary a tracepoint BPF program from having no instructions to the maximum possible instructions that the current verifier allows. As the test is being carried out in Linux kernel 6.0.2, privileged BPF programs have no instruction limit and only need to be less than 1M verified instructions[53]. For the test, we wrote a simple counter program incremented in a loop about 100k times which led to reaching the maximum verifier limit. Running this program will provide us the runtime for a BPF program having the worst-possible runtime due to non-helper call instructions alone that won't get stubbed out even after the termination procedure. Figure 5.3 shows the results.

3. **Helper Runtime** Helper functions can show a range of runtime due to reasons

| Component | Runtime (ns) Min | Max |
|---|---|---|
| Termination handler | 1255144 | 1255144 |
| Stub runtime | 82 | 685333 |
| Helper runtime | 0 | 4500 |
| **Termination time** | 1255226 | 1944977 |

Table 5.3: The approximated range of termination delay is between 1.2-1.9 msec

such as different function arguments, different execution environments, etc as described in § 3.2.2. Figure 3.2 showed possible runtimes for several helper functions. Thus, we take the values previously evaluated for the minimum and maximum helper runtime in Table 5.3.

Summing the values obtained from the three components, we get a range of termination time as shown in Table 5.3 which suggests the maximum termination time is $\approx 1.9$ *msec*. Due to the nature of values obtained for the Termination handler, the overall range obtained for termination time is only indicative.

We further validate the end-to-end termination delay by issuing a termination request to a BPF program. In the setup, we attach a BPF program containing a helper call to fetch the current NUMA node ID and attach it as a tracepoint program to our empty syscall described before. On another available CPU, we issue the termination syscall providing the target BPF program ID which will go through the different steps for the proposed termination. This experiment gave a termination delay of *1584629 ns* which is within the approximate predicted range for termination delay obtained in Table 5.3.

# Chapter 6

# Related Work

Extensibility has been studied widely in systems research with the target to provide most flexible interface that allows extending the OS at different granularity. Historically, the target use-cases include customizing eviction policies for file cache, signal delivery, etc which needed exposing low-level kernel interfaces to extensions[50]. Broadly, the applications can be categorized into: enforcing policies, and providing mechanism.

eBPF(formerly acronym for extended Berkeley Packet Filter) is the modern extension framework in Linux kernel that provides access to low-level hooks in the kernel where a user-provided code can be injected. eBPF is based on top of the former BPF[41] which was introduced as an advancement to the iptables[6] based packet-filtering. BPF provides safety properties of all the installed extensions using a static verifier. BPF extensions are being used in use-cases like load balancing [51], profiling [2] and security enforcement [3].

## 6.1 BPF Performance

Performance prediction of Network Functions is an active area of interest due to rise of Network Function Virtualization (NFVs) on commercial middleboxes. In [28, 29, 44, 47], the authors use symbolic execution to generate a performance profile that can predict cost for any given workload. But their work depends on annotations of all the involved data structures regarding memory read-write costs based on which the overall profile is generated. While the simplicity of NFs allow such annotations, similar annotations are infeasible for all BPF helpers because of the large number of kernel data structures which interact with BPF helpers at any given point [38]. Network Functions rely on optimizations like process co-location and cache partitioning to reduce hardware contention. [21, 40] have attempted to model the hardware properties like cache occupancy and memory bandwidth to predict additional latency that would be caused for each new co-located competitors in a NFV environment. However, contentions in a general system can have multiple other sources such as CPU memory capacity, storage bandwidth, CPU-socket interconnect, etc which makes the modelling much more difficult and error prone.

Worse Case Execution Time (WCET) is a widely used performance metric in real-time systems [57]. Due to the restrictive programming environment of eBPFs, estimating the runtime of eBPF is very similar to estimating WCET of real-time systems. Worse case estimation problem has been broadly dealt using two classes of approaches: static analysis, and dynamic measurements. The static analysis depends on Control Flow Analysis (CFA) to determine loop bounds and recursion depths to provide an upper bound on WCET [14], while dynamic measurements aim to provide estimates closer to the hardware level by performing end-to-end tests of code sub-components [18].

Attempts to use static analysis for WCET estimation in single-core architecture have followed

several approaches such as off-line prediction [16], Genetic Algorithms [7], Integer Linear Programming [36, 58] or statistical methods like the Extreme Value Theory [23, 26, 37]. But with the onset of multi-core architectures, estimating WCET faces the trade-off between analysis of all possible influence due to shared hardware resources, and making simplified assumptions to reduce complexity [18, 39]. Due to modern processor architecture like caches and pipelines, the combined runtime of two sequential pieces of code can greatly differ from the sum of their individual runtime due to the execution state generated by the former program.

Dynamic approach such as fuzzing [11, 35, 55] operates by generating new test cases by mutating seed inputs to discover new paths during execution. While the efficiency of dynamic analysis is improving with the latest contributions, they do not guarantee to visit every hidden branch of the code. The community has, therefore, evolved with hybrid solutions [10, 18, 31, 32, 34, 56] which combines statically computed Control-Flow Graph with execution time collected through dynamic measurements to provide more precise estimation results.

## 6.2 BPF Termination

Termination of extensions in general has been advocated in works like VINO[50] where the authors considered the need for safe preemption of extensions as a key factor into their design. While BPF, due to its imposed limitation on number of iterations and back-edges, proves eventual termination, there is currently no accepted solution to deal with long-running extensions that needs to be terminated. In [22], the author has proposed pre-generating stack frames for all points of a BPF program where an exception could be thrown. However, supporting abrupt termination request that could arise during any instruction of a program will lead to large memory footprint due to number of stack frames needed scaling with the

number of instructions in a BPF program.

Industry standards techniques for handling exceptions/termination in C++ and JVM have faced unique challenges. In [12, 15, 45] the authors find/study various bugs induced due to incorrect exception handling in C++. [45] also explains how getting another exception while handling one further complicates the exception handling which java solves by using finalizers. [46] proposed a stack based Exception state solution to avoid bloating the binary with complex unwind logic and instead use a run-time type information to be pushed to every function as an argument. While this approach reduced the size of binary by considerable margin, it had a negative affect on a no-exception path due to the additional argument and associated checks. As C++ unwinding depends on the DWARF debugging format[4], bugs in the DWARF unwinding tables can lead to incorrect stack cleanup and expoits[43]. Works like [9] have tried to reduce bugs in a C++ binary's DWARF unwind tables by cross-checking them with the generated binary after compilation.

In Java's JVM, several works in the literature have approached the issue of termination. In [48], the authors propose using a simple flag check to implement soft termination in JVM for resource management. The intended use case for termination were sysadmins or system resource monitors that observed a program to exceed the allotted resources such as CPU, memory, network bandwidth, etc. However, despite its simplicity, this solution imposes a flag check overhead even when termination is not needed while also affecting loops the worst because of the flag check happening once every iteration. In [8], the authors suggested creating a virtual red-line between user-space and kenel tasks for termination. Whenever terminate is requested, the kernel with wait till the program reaches back into the user-space boundary. This model is similar to the UNIX implementation where a signal to kill a process is queued until the process execution returns from the kernel context. In [13], the authors propose graceful termination of a group of related processes, called as process

networks for Java-Communication Sequential Processes. They use a concept of poisoning by passing termination object to each interacting process through exceptions and starts the cleanup/unwinding procedure while also poisoning all dependent child processes to the current process.

**On-Stack Replacement:** Modern JIT compilers use On-Stack Replacement (OSR) techniques to switch a program between different versions to suit varied interests. Common use cases include, optimizing functions for lower start-up latency but replacing it with a code optimized for higher throughput if profiling indicates a higher than expected number of iterations. Another use case is to de-optimize a program which the compiler had speculatively optimized previously but ends up realising the assumptions made during optimization doesn't hold anymore, and thus has to replace the program with a safer version[19].

# Chapter 7

# Future Work & Conclusion

In the final chapter of this thesis, we describe some next steps that can be pursued to either circumvent the limitations of the mechanisms discussed until now or projects that are enabled due to the proposed solutions. For the next steps, we describe three possible directions: 1) Framing policies for BPF Orchestrators; 2) The need for a global view; and 3) Making BPF programs atomic. Lastly, we sum up this thesis in the Conclusion section.

## 7.1 Framing policies for BPF Orchestrators

While motivating the need for a BPF Runtime Estimator, we described BPF Orchestrator as one of the clients who can make use of the Estimator to make better policies. Our definition of a BPF Orchestrator is a control plane that allows an administrator to configure policies that can restrict the capabilities of a user trying to load BPF programs into the system. During the motivation in chapter 2, we defined two types of policies: role-based admission control policies, and runtime-overhead policies, where the proposed BPF Runtime Estimator can facilitate the latter. While role-based policy is self-explanatory, making policy based on

the predicted runtime of BPF programs needs more clarification. Runtime-based policy can be about restricting the latency induced in critical paths of a system by low-privileged users, for example. However, such a policy will require a threshold value corresponding to each possible call path in the system where a frequently used call path will have a strict threshold whereas rare paths will have much-relaxed latency constraints.

An immediate challenge is to accurately and efficiently find the right configurations that align with a given workload. For example, a server with excessive memory usage should not perform heavy tracing operations on memory-allocating functions. Currently, an operator is trusted to understand their system well, and their workloads even better, such that they can correctly perform configurations that will best suit the needs of the system. Having an automated solution to generate the best configuration will not only remove the guesswork but also may end up realizing un-intuitive config values which are much more efficient than a human expect.

## 7.2 The need for a global view

Making claims about the performance of individual extensions alone is not sufficient. A big chunk of high runtime issues arise due to unintentional interaction between multiple BPF programs. For example, a BPF program might end up triggering another BPF program in the process of doing its operation (such as making a helper call). Nesting or chaining of BPF programs can add unwanted latency on a fast path which an operator might want to carefully reason about. Not being aware of such a possibility of nesting can lead to even more severe performance degradation and failed SLA agreements. We therefore need a system-wide view where we can reason about the overall performance impact due to different BPF programs taking into account the individual contribution of each program.

The requirement is not limited to getting a better image of performance impact due to BPF programs. With the growing adoption of BPF, there has been an increase in the number of deadlock scenarios discovered by developers and automated testing scripts where two nested BPF programs were found to be deadlocked due to a shared lock being attempted to acquire twice. As the current BPF verifier's analysis is only limited to the granularity of a single BPF program, the BPF subsystem needs a makeover to have a better understanding of system-wide BPF-to-BPF and BPF-kernel interactions.

Possible solutions include expanding the scope of the BPF verifier to be aware of control flow changes in the kernel due to a newly attached BPF program and look for new classes of issues such as deadlocks, etc along with the existing issue of long-runtimes. While existing BPF Orchestrators can be modified to make them call-graph aware, being a userspace application will limit the visibility of such an orchestrator in terms of kernel objects or function pointers, which will be better available for an in-kernel "super" verifier.

## 7.3 Making BPF programs atomic

With the introduction of BPF termination and ongoing work to include exception mechanisms, a BPF invocation can now be terminated before all instructions are executed. Until now, a BPF program had an unsaid assumption to run as an atomic unit, where a programmer expected all the state modifications done within a program to always be complete or not be done at all. However, due to termination, the assumption is no more valid and the program can reach inconsistent states where, say, some map writes were executed, while others were skipped due to termination. For example, a BPF program taking metrics at an *XDP* or a *tc* hook which uses two different map writes, one for packet count and the other for packet size count, will become incorrect if the packet count gets updated, while the size

count couldn't be written due to termination. Therefore, some mechanism to roll back to a consistent state is necessary for maintaining the correctness of existing data[49].

## 7.4 Conclusion

In this work, we investigated BPF programs along the axis of their runtimes and performance impact. We found the helper function interface to be the major culprit behind both issues. To propose a solution, we studied the runtime variability of some of the long-running helper functions to create an estimator that combines the verifier's static analysis with the dynamic measurements for helper functions. This runtime estimator is well suited as a preventive solution to stop potentially bad-performing BPF programs from installing into the system. For cases when obtaining a deterministic performance estimate is not possible, either due to dynamic parameters or effects due to the runtime environment, we proposed a termination mechanism to kill violating BPF programs. The termination mechanism involved creating a short-circuited copy of a BPF program which is used to quickly finish an ongoing execution. In our evaluations, we found the runtime estimator's results to be very broad to be used to design strict policies, however, when complemented with the termination mechanism, performance-based policies can still be enforced in case a BPF program is found to show unexpectedly worse performance. The fast-path termination solution showed zero overhead on a no-termination case which aligns with the design goal, thus proving that using the proposed solution will bring back the guarantee of termination to BPF without paying any additional cost. For future work, we see a need for a wider view in the verifier to better handle complex cases like program nesting, while a need for making BPF execution transactional will reinforce the atomicity assumptions using which almost all BPF projects are operating today.

# Bibliography


[1] BPF architecture. https://docs.cilium.io/en/latest/bpf/architecture/#instruction-set, . Accessed: 2024-05-23.

[2] Bpftrace. https://bpftrace.org/, . Accessed: 2024-05-23.

[3] Tertragon:eBPF-based security observability and runtime enforcement. https://github.com/cilium/tetragon, . Accessed: 2024-05-23.

[4] Dwarf debugging standard website. https://dwarfstd.org/. Accessed: 2023-06-08.

[5] Bpf_map_type_hash, with percpu and lru variants. https://www.kernel.org/doc/html/next/bpf/map_hash.html. Accessed: 2024-05-23.

[6] The netfilter.org project. https://www.netfilter.org/. Accessed: 2023-06-08.

[7] Jaswinder Ahluwalia, Ingolf H Krüger, Walter Phillips, and Michael Meisinger. Model-based run-time monitoring of end-to-end deadlines. In *Proceedings of the 5th ACM international conference on Embedded software*, pages 100–109, 2005.

[8] Godmar Back and Wilson C Hsieh. The kaffeos java runtime system. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(4):583–630, 2005.

[9] Théophile Bastian, Stephen Kell, and Francesco Zappa Nardelli. Reliable and fast dwarf-based stack unwinding. *Proceedings of the ACM on Programming Languages*, 3 (OOPSLA):1–24, 2019.

[10] G. Bernat, A. Colin, and S.M. Petters. Wcet analysis of probabilistic hard real-time systems. In *23rd IEEE Real-Time Systems Symposium, 2002. RTSS 2002.*, pages 279–288, 2002. doi: 10.1109/REAL.2002.1181582.

[11] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1032–1043, 2016.

[12] Kirsten Bradley and Michael Godfrey. A study on the effects of exception usage in open-source c++ systems. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 1–11. IEEE, 2019.

[13] Jan F Broenink, HW Roebbers, and JPE Sunter. *Communicating Process Architectures 2005: WoTUG-28*, volume 63. IOS Press, 2005.

[14] Giorgio C Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*, volume 24. Springer Science & Business Media, 2011.

[15] Tom Cargill. Exception handling: A false sense of security. In *C++ gems*, pages 423–431. 1996.

[16] Antoine Colin and Isabelle Puaut. Worst-case execution time analysis of the rtems real-time operating system. In *Proceedings 13th Euromicro Conference on Real-Time Systems*, pages 191–198. IEEE, 2001.

[17] Jonathan Corbet. Toward signed BPF programs. https://lwn.net/Articles/853489/. Accessed: 2024-05-23.

[18] Robert I Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM computing surveys (CSUR)*, 43(4):1–44, 2011.

[19] Daniele Cono D'Elia and Camil Demetrescu. Flexible on-stack replacement in llvm. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pages 250–260, 2016.

[20] Luca Deri, Samuele Sabella, Simone Mainardi, Pierpaolo Degano, and Roberto Zunino. Combining system visibility and security using ebpf. In *ITASEC*, volume 2315, 2019.

[21] Mihai Dobrescu, Katerina Argyraki, and Sylvia Ratnasamy. Toward predictable performance in software packet-processing platforms. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 141–154, 2012.

[22] Kumar Kartikeya Dwivedi. [rfc patch v1 00/14] exceptions - resource cleanup. https://lore.kernel.org/all/20240201042109.1150490-1-memxor@gmail.com/. Accessed: 2024-05-23.

[23] Stewart Edgar and Alan Burns. Statistical analysis of wcet for scheduling. In *Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001)(Cat. No. 01PR1420)*, pages 215–224. IEEE, 2001.

[24] Brendan Gregg. BPF performance analysis at netflix. https://d1.awsstatic.com/events/reinvent/2019/REPEAT_1_BPF_performance_analysis_at_Netflix_OPN303-R1.pdf. Accessed: 2024-05-23.

[25] Brendan Gregg. *BPF performance tools*. Addison-Wesley Professional, 2019.

[26] Jeffery Hansen, Scott Hissam, and Gabriel A Moreno. Statistical-based wcet estimation and validation. In *9th international workshop on worst-case execution time analysis (WCET'09)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2009.

[27] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The express data path: Fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th international conference on emerging networking experiments and technologies*, pages 54–66, 2018.

[28] Rishabh Iyer, Luis Pedrosa, Arseniy Zaostrovnykh, Solal Pirelli, Katerina Argyraki, and George Candea. Performance contracts for software network functions. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 517–530, 2019.

[29] Rishabh Iyer, Katerina Argyraki, and George Candea. Performance interfaces for network functions. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 567–584, 2022.

[30] Jinghao Jia, YiFei Zhu, Dan Williams, Andrea Arcangeli, Claudio Canella, Hubertus Franke, Tobin Feldman-Fitzthum, Dimitrios Skarlatos, Daniel Gruss, and Tianyin Xu. Programmable system call security with ebpf. *arXiv preprint arXiv:2302.10366*, 2023.

[31] Daniel Kästner, Markus Pister, Simon Wegener, and Christian Ferdinand. Timeweaver: A tool for hybrid worst-case execution time analysis. In *19th International Workshop on Worst-Case Execution Time Analysis (WCET 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[32] Raimund Kirner, Ingomar Wenzel, Bernhard Rieder, and Peter Puschner. Using measurements as a complement to static worst-case execution time analysis. *Intelligent Systems at the Service of Mankind*, 2(8):20, 2005.

[33] Hsuan-Chi Kuo, Kai-Hsun Chen, Yicheng Lu, Dan Williams, Sibin Mohan, and Tianyin Xu. Verified programs can party: optimizing kernel extensions via post-verification

merging. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 283–299, 2022.

[34] Stephen Law, Mike Bennett, Stuart Hutchesson, Ivan Ellis, Guillem Bernat, Antoine Colin, and Andrew Coombes. Effective worst-case execution time analysis of do178c level a software. *Ada User Journal*, 36(3), 2015.

[35] Xuan-Bach D Le, Corina Pasareanu, Rohan Padhye, David Lo, Willem Visser, and Koushik Sen. Saffron: Adaptive grammar-based fuzzing for worst-case analysis. *ACM SIGSOFT Software Engineering Notes*, 44(4):14–14, 2021.

[36] Björn Lisper. Fully automatic, parametric worst-case execution time analysis. *WCET*, 3:77–80, 2003.

[37] Yue Lu, Thomas Nolte, Iain Bate, and Liliana Cucu-Grosjean. A new way about using statistical analysis of worst-case execution times. *ACM SIGBED Review*, 8(3):11–14, 2011.

[38] Alan Maguire. A zoological guide to kernel data structures. https://blogs.oracle.com/linux/post/a-zoological-guide-to-kernel-data-structures. Accessed: 2024-05-23.

[39] Claire Maiza, Hamza Rihani, Juan M Rivas, Joël Goossens, Sebastian Altmeyer, and Robert I Davis. A survey of timing verification techniques for multi-core real-time systems. *ACM Computing Surveys (CSUR)*, 52(3):1–38, 2019.

[40] Antonis Manousis, Rahul Anand Sharma, Vyas Sekar, and Justine Sherry. Contention-aware performance prediction for virtualized network functions. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on*

*the applications, technologies, architectures, and protocols for computer communication*, pages 270–282, 2020.

[41] Steven McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *USENIX winter*, volume 46, 1993.

[42] Andrii Nakryiko. BPF open-coded iterators. https://lwn.net/Articles/925751/. Accessed: 2024-05-23.

[43] James Oakley. Exploiting the hard-working dwarf: Trojan and exploit techniques with no native executable code. In *5th USENIX Workshop on Offensive Technologies (WOOT 11)*, 2011.

[44] Luis Pedrosa, Rishabh Iyer, Arseniy Zaostrovnykh, Jonas Fietz, and Katerina Argyraki. Automated synthesis of adversarial workloads for network functions. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 372–385, 2018.

[45] Prakash Prabhu, Naoto Maeda, Gogul Balakrishnan, Franjo Ivančić, and Aarti Gupta. Interprocedural exception analysis for c++. In *European Conference on Object-Oriented Programming*, pages 583–608. Springer, 2011.

[46] James Renwick, Tom Spink, and Björn Franke. Low-cost deterministic c++ exceptions for embedded systems. In *Proceedings of the 28th International Conference on Compiler Construction*, pages 76–86, 2019.

[47] Daniele Rogora, Antonio Carzaniga, Amer Diwan, Matthias Hauswirth, and Robert Soulé. Analyzing system performance with probabilistic performance annotations. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–14, 2020.

[48] Algis Rudys and Dan S Wallach. Termination in language-based systems. *ACM Transactions on Information and System Security (TISSEC)*, 5(2):138–168, 2002.

[49] Algis Rudys and Dan S Wallach. Transactional rollback for language-based systems. In *Proceedings International Conference on Dependable Systems and Networks*, pages 439–448. IEEE, 2002.

[50] 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.

[51] Nikita Shirokov and Ranjeeth Dasineni. Open-sourcing katran, a scalable network load balancer. https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer/. Accessed: 2024-05-23.

[52] Alexei Starovoitov. BPF: introduce function calls. https://lwn.net/Articles/741773/, . Accessed: 2024-05-23.

[53] Alexei Starovoitov. Kernel mailing list patch for increasing bpf instruction limit to 1m. https://patchwork.ozlabs.org/project/netdev/patch/20190402042749.3670015-7-ast@kernel.org/, . Accessed: 2024-04-18.

[54] Patrick Thibodeau. Booted up in 1993, this server still runs — but not for much longer. https://www.computerworld.com/article/1673071/booted-up-in-1993-this-server-still-runs-but-not-for-much-longer-2.html. Accessed: 2024-05-23.

[55] Jiayi Wei, Jia Chen, Yu Feng, Kostas Ferles, and Isil Dillig. Singularity: Pattern fuzzing for worst case complexity. In *Proceedings of the 2018 26th ACM Joint Meeting*

*on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 213–223, 2018.

[56] Ingomar Wenzel, Raimund Kirner, Bernhard Rieder, and Peter Puschner. Measurement-based worst-case execution time analysis. In *Third IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS'05)*, pages 7–10. IEEE, 2005.

[57] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):1–53, 2008.

[58] Wei Zhang and Jun Yan. Accurately estimating worst-case execution time for multi-core processors with shared direct-mapped instruction caches. In *2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 455–463. IEEE, 2009.