Enabling BPF Runtime policies for better BPF management

Raj Sahu  
Virginia Tech  
Blacksburg, VA, USA  
raj.sahu@vt.edu

Dan Williams  
Virginia Tech  
Blacksburg, VA, USA  
djwillia@vt.edu

ABSTRACT

As eBPF increasingly and rapidly gains popularity for observability, performance, troubleshooting, and security in production environments, a problem is emerging around how to manage the multitude of BPF programs installed into the kernel. Operators of distributed systems are already beginning to use BPF-orchestration frameworks with which they can set load and access policies for who can load BPF programs and access their resultant data. However, other than a guarantee of eventual termination, operators currently have little to no visibility into the runtime characteristics of BPF programs and thus cannot set policies that ensure their systems still meet crucial performance targets when instrumented with BPF programs. In this paper, we propose that having a runtime estimate will enable better policies that will govern the allowed latency in critical paths. Our key insight is to leverage the existing architecture within the verifier to statically track the runtime cost of all possible branches. Along with dynamically determined runtime estimates for helper functions and knowledge of loop-based helpers’ effects on control flow, we generate an accurate—although broad—range estimate for making runtime policy decisions. We further discuss some of the limitations of this approach, particularly in the case of broad estimate ranges as well as complementary tools for BPF runtime management.

CCS CONCEPTS

• General and reference → Performance;  • Software and its engineering → Software performance; Virtual machines; Real-time systems software; Operating systems;

KEYWORDS

eBPF, Policy, Orchestration

ACM Reference Format:


1 INTRODUCTION

eBPF is a special instruction set virtual machine which is gaining popularity due to its ability to directly load a program into the Linux kernel while guaranteeing safety. With emerging use cases, production servers are increasingly becoming dependent on a variety of BPF programs [1] for managing load [2], profiling [3], security [4], etc. As a result, the community has developed BPF management/orchestration frameworks [5, 6] which system administrators can use to get better control over the BPF programs running in their system. A BPF-orchestrator acts as a control plane and provides APIs for the users to interact with BPF programs in the system. The interface allows a sysadmin to configure role-based policies that will allow users to load BPF programs based on their roles and privilege. These policies can be used to secure important nodes of a cluster, prevent a low-privileged user from loading to a critical hook point of the kernel, reduce the list of loaded BPF programs accessible to a user for data collection, etc.

Even though these policies can greatly reduce unexpected behavior by associating users with the least possible privilege, there currently exists no reliable mechanism to determine how costly these BPF programs can be to the system in the worst case. A slow-running BPF program can prove to be bottleneck at any frequently used call path within an operating system [7]. Thus 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.

Existing works have shown symbolic execution, similar to what is done by the BPF verifier, can obtain a promising level of performance prediction for Network Functions (NFs) [8, 9]. We propose a Runtime Estimator which is a hybrid of static and dynamic analysis that is more generally applicable to any BPF program. Using dynamic measurements and static analysis, the modified verifier generates an overall estimate of the best-case to worst-case runtime of a BPF program. We test the proposed solution against some sample eBPF programs from the Linux kernel source code and observe the estimates to be aligned to actual runtimes.

We also observe the runtime estimates to be very broad in some cases. Preliminary investigation reveals that some of the helper functions, on which BPF programs are usually dependent, can show high variance due to I/O operations, argument-dependence, or lock-contention. We intend to investigate ways to capture this variability in our estimates for future work.

1.1 Contributions

The key contributions made in this work are:

• We identify runtime estimation as a critical requirement for emerging BPF policies.
• We propose a BPF runtime estimation mechanism to determine the performance cost of any given BPF program during verification.
After successful verification, the admin attaches the program to a hook point. An attached program has access to helper functions which form an interface to kernel functions and provide common utilities like modifying network packets, injecting faults, generating random numbers, etc. A userspace program can interact with the BPF programs using map data structures to share information across the kernel-userspace interface.

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 1 gives an overview of how BPF-orchestrators work in a cluster environment.

2.1 Background

eBPF is an in-kernel extension framework that allows an admin to dynamically load extension programs into the Linux kernel without needing a system reboot. Unlike inherently unsafe kernel modules, eBPF programs need to pass a verifier which performs static analysis on the BPF bytecode to ensure memory safety and termination. After successful verification, the admin attaches the program to a hook point. An attached program has access to helper functions which form an interface to kernel functions and provide common utilities like modifying network packets, injecting faults, generating random numbers, etc. A userspace program can interact with the BPF programs using map data structures to share information across the kernel-userspace interface.

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 1 gives an overview of how BPF-orchestrators work in a cluster environment.

2.2 Runtime Policies

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.

Currently, the best guarantee provided by the eBPF verifier is that of eventual termination, but this guarantee itself does not indicate how rapidly a program will terminate. We wrote a simple eBPF program using nested loops.

```c
main():
    prog_1 (void):
        bpf_printk ("foo")
    prog_2 (void):
        bpf_loop (1000, prog_1)
    prog_n (void):
        bpf_loop (1000, simple)

Listing 1: Pseudo-code showing a sample long running BPF program using nested loops.
```

like Kubernetes. An administrator configures load and access policies. The load policies can include signature validation[10], 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 pods. 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.

2.1 Background

In this section we provide background about BPF management and highlight a gap around runtime policies.

2.2 Runtime Policies

In this section we present relevant background information and describe the motivation for this work. Section §3 describes the design for the proposed solution followed by evaluation in Section §4. Section §5 highlights key discussion points observed during this work. We end with related work and a concluding summary in Section §6 and Section §7 respectively.
program using the \texttt{bpf\_loop} helper which could run indefinitely despite being validated by the verifier. Listing 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.

A naive approach based on dynamic benchmarking using fuzzers or the \texttt{bpf\_tool 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 soundness problem as certain aspects of a program like function pointers and function arguments are only known during runtime. However, static analysis has not been investigated in the context of eBPF runtime.

### 2.3 Challenges

While the restricted complexity of eBPF [11] 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[12] 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 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 \texttt{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 [13], the proposed solution will need to consider the influence of these helpers on the runtime estimation.

In the next section, we address the above challenges and discuss how the proposed solution tackles them.

### 3 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 theCFG generated by the verifier.

#### 3.1 The Runtime Estimator

Figure 2 shows the modified architecture where the verifier now includes a Runtime Estimator. 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. For obtaining the estimates, we use BPF samples from the Linux kernel repository with the required helper and attach it to the intended hook. The samples were modified such that each helper is invoked 1000 times to report the best, average, and worst case execution times. The modified sample BPF programs were triggered for 10 different times to capture performance across varying system load. For map based helpers, we used the LRU hashmap (BPF\_MAP\_TYPE\_LRU\_HASH). For measuring execution time, we used the \texttt{bpf\_ktime\_get\_ns} helper and assume it’s variance to be negligible.

Figure 3 shows the runtime estimates obtained for 31 helper functions. The length of the bars describes how some helpers like \texttt{bpf\_tcp\_sock} show a tight bound on runtimes while helpers like \texttt{bpf\_map\_update\_elem} and \texttt{bpf\_probe\_read\_user\_str} show a high variation due to dependency on an argument or because of lock contention, which we will further discuss in §5. 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 worse 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.

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 as discussed in Section 5.

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 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 bpfIter, 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.

4 EVALUATION

In this section, we use the observations made during helper-runtime calculations(§3) 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 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 traceex1, 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). 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) 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 2) with nested bpf_loop helper calls. We use this simple program to demonstrate a rare branch which can be easily missed in dynamic benchmarking.

Listing 2: Pseudo-code showing nested loops with branching in a BPF program.
When this program was attached and triggered, we observed an increase in the performance of eBPF programs. We have reduced the ambiguity of the expected runtime range of a BPF program by taking locks before updating values in a BPF map, since the worst case belongs to the branch containing nested loops.

Argument-dependent variance: We identified two different helper function classes based on source of lock-taking helper functions. Upon further investigation, we found the source of variation and then raised other errors to become more infrequent over time.

Table 1: Expected and actual runtime of eBPF samples

<table>
<thead>
<tr>
<th>Sample Program</th>
<th>Expected Runtime</th>
<th>Actual Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>trace1</td>
<td>192-2660</td>
<td>1011</td>
</tr>
<tr>
<td>trace2</td>
<td>48-4028</td>
<td>88</td>
</tr>
<tr>
<td>trace3</td>
<td>249-2040</td>
<td>468</td>
</tr>
<tr>
<td>trace4</td>
<td>48-3454</td>
<td>279</td>
</tr>
<tr>
<td>trace5</td>
<td>48-800</td>
<td>123</td>
</tr>
<tr>
<td>sockex1</td>
<td>86-590</td>
<td>225</td>
</tr>
<tr>
<td>sockex2</td>
<td>86-3424</td>
<td>551</td>
</tr>
<tr>
<td>sockex3</td>
<td>86-4274</td>
<td>123</td>
</tr>
<tr>
<td>test_current_task_under_cgroup</td>
<td>315-3224</td>
<td>753</td>
</tr>
<tr>
<td>test_probe_write_user</td>
<td>48-2450</td>
<td>761</td>
</tr>
<tr>
<td>trace_event</td>
<td>171-7878</td>
<td>6252</td>
</tr>
<tr>
<td>tcp_baserttt</td>
<td>171-2220</td>
<td>1039</td>
</tr>
<tr>
<td>tcp_dumpstats</td>
<td>57-3470</td>
<td>2277</td>
</tr>
</tbody>
</table>

Table 1: Expected and actual runtime of eBPF samples

Dynamic runtime mechanisms: As our static analysis based solution shows high variation due to factors discussed above, we consider the situation in which an operator may want to make use of aggressive policies or average case policies that occasionally turn out to be incorrect. In such a case, dynamic runtime strategies such as runtime termination can enforce strict timers on the maximum allowed runtime for BPF programs. For example, if the system detects a BPF program has violated its permitted runtime constraints, then the system can take appropriate steps via the use of control mechanisms.

Runtime termination of eBPF programs is not trivial. eBPF programs run in the kernel, but kernel code is not safe to abruptly terminate, as it may hold locks or references to kernel objects that must first be released. Indeed, some of the eBPF verifier checks ensure that locks and kernel references are released on every code path, assuming that the program will terminate (as is also guaranteed by the verifier). As discussed in Section 2.2, termination may not be timely. We are actively working on dynamic termination mechanisms to unwind early terminating eBPF programs, thereby releasing all locks and kernel resources. We believe a runtime mechanism will greatly compliment an operator’s control over the latency induced by BPF programs.

5 DISCUSSION & FUTURE WORK

In this section, we further discuss the implications of the helper function runtime variation that we observed and then raise other limitations and opportunities for runtime eBPF management.

Dealing with helper runtime variability: By using dynamic measurements of helpers instead of that for the whole BPF program, we have reduced the ambiguity of the expected runtime range of a BPF program from extension program-level granularity to helper-level granularity. However, as some of the most complex helper functions have a call graph more than 20 nodes deep, the worst-case runtime of a helper can be significantly different from the average case observed through dynamic measurements.

Upon further investigation, we found the source of variation and identified two different helper function classes based on source of variance: argument-dependent and resource-dependent.

Argument-dependent helpers such as the bpf_get_stackid and the bpf_perf_event_output helper show variation in runtimes with change in arguments. If the argument is determinable statically or at load time (e.g., the particular hook point that the program is loaded at may dictate a fixed context argument to a helper), we believe that our runtime estimator should be able to use this relationship to improve our estimates.

On the other hand, helpers like bpf_map_update_elem operate by taking locks before updating values in a BPF map, since the runtime could be shared between multiple eBPF programs. Depending on the number of threads and frequency of map access (which are generally not statically determinable), we expect to encounter increased overhead due to lock contention. In our tests, we observed that the average runtime of bpf_map_update_elem increased by 2.5x and worst case runtime increased by ~ 4x when run concurrently in 2 CPUs. Accounting for latency due to lock contention and cache coherence is generally a hard problem with active research and we plan to study how the worst-case runtimes are approximated in multi-core architectures. While not the case for all locks or contended kernel resources, for some resources, we may be able to determine how much contention to expect on a lock and provide a better estimate. For example, a map may only be referenced by eBPF programs, the number and concurrency of which may be known to the BPF orchestrator at load time. Finally, we are also interested in whether lock-taking helper functions can be restructured/simplified for more deterministic runtimes.

6 RELATED WORKS

Performance prediction of Network Functions is an active area of interest due to rise of Network Function Virtualization (NFVs) on commercial middleboxes. In [8, 9, 15, 16], 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.
structures which interact with BPF helpers at any given point [17]. Network Functions rely on optimizations like process co-location and cache partitioning to reduce hardware contention. [18, 19] 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, contentsions 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 [20]. 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 [21], while dynamic measurements aim to provide estimates closer to the hardware level by performing end-to-end tests of code sub-components [22].

Attempts to use static analysis for WCET estimation in single-core architecture have followed several approaches such as off-line prediction [23], Genetic Algorithms [24], Integer Linear Programming [25, 26] or statistical methods like the Extreme Value Theory [27–29]. 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 [22, 29]. 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 [31–33] 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 [22, 34–38] which combines statically computed Control-Flow Graph with execution time collected through dynamic measurements to provide more precise estimation results.

7 SUMMARY
In this work, we identified and highlighted the importance of BPF runtime estimation for making stronger BPF-management policies. Our proposed solution used a hybrid of static and dynamic analysis to provide runtime estimates of BPF programs at verification time, which can now be used by userspace BPF-orchestration frameworks to limit latency in critical paths of the kernel. Our evaluations suggest lowering the variance of argument-dependent or lock-taking helper function to obtain tighter bounds of expected runtimes. In future work, we plan to analyse argument-dependent and lock-taking helpers to better understand their variation with function parameters and contention, respectively.

Acknowledgement
This work is supported in part by NSF grant CNS-2236966.

References
Enabling BPF Runtime policies for better BPF management


