Eliminating eBPF Tracing Overhead on Untraced Processes

Milo Craun
Virginia Tech
Blacksburg, VA, USA
miloc@vt.edu

Khizar Hussain
Virginia Tech
Blacksburg, VA, USA
khizar@vt.edu

Uddhav Gautam
Virginia Tech
Blacksburg, VA, USA
upgautam@vt.edu

Zhengjie Ji
Virginia Tech
Blacksburg, VA, USA
zhengjie@vt.edu

Tanuj Rao
Virginia Tech
Blacksburg, VA, USA
tansanrao@vt.edu

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

ABSTRACT

Current eBPF-based kernel extensions affect entire systems, and are coarse-grained. For some use cases, like tracing, operators are more interested in tracing a subset of processes (e.g., belonging to a container) rather than all processes. While overhead from tracing is expected for targeted processes, we find *untraced processes*—those that are not the target of tracing—also incur performance overhead. To better understand this overhead, we identify and explore three techniques for per-process filtering for eBPF: post-eBPF, in-eBPF, and pre-eBPF filtering, finding that all three approaches result in excessive overhead on untraced processes. Finally, we propose a system that allows for zero-untraced-overhead per-process eBPF tracing by modifying kernel virtual memory mappings to present *per-process kernel views*, effectively enabling untraced processes to execute on the kernel as if no eBPF programs are attached.

CCS CONCEPTS

• **Software and its engineering** → **Operating systems**; *Software testing and debugging*; *Software performance*;

KEYWORDS

eBPF, dynamic tracing, tracing overhead, copy-on-write

ACM Reference Format:
Milo Craun, Khizar Hussain, Uddhav Gautam, Zhengjie Ji, Tanuj Rao, and Dan Williams. 2024. Eliminating eBPF Tracing Overhead on Untraced Processes. In *Workshop on eBPF and Kernel Extensions (eBPF '24), August 4–8, 2024, Sydney, NSW, Australia.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3672197.3673431

1 INTRODUCTION

eBPF is a Linux kernel subsystem that allows for safe kernel extensions to be loaded into the kernel. eBPF programs are used for a variety of use cases from packet filtering [5, 15, 18, 24, 37, 38, 43, 51, 55], to profiling [3, 10, 23, 51], to application acceleration [20, 34, 39, 59, 60], to system tracing [4] and security [2, 13, 14, 19, 26, 31]. Extensions attach to a set of *hookpoints*, which cause the extension to be executed whenever a certain system event occurs (e.g. system call). eBPF's event based model of execution is especially useful for tracing use cases. However, the current implementation of eBPF does not natively support per-process tracing, relying instead on other methods.

One method to achieve per-process tracing, which we term *post-eBPF filtering*, is to capture all events, and then filter for relevant processes after data collection. Another method, *in-eBPF filtering*, used by bpftrace [4], filters for processes of interest inside eBPF programs. Per-process tracing can also be achieved by checking the PID before eBPF program execution through kernel modification, in a technique we refer to as *pre-eBPF filtering*. We explore the costs of all three approaches and find that they place significant overheads on untraced processes, referred to as *untraced overhead*. We conclude that a new system is needed to enable zero untraced overhead per-process tracing.

We propose a novel approach which provides individual processes their own view of kernel hookpoints. When an eBPF program is attached to a hookpoint, a *kernel view manager* essentially performs copy-on-write to attach the eBPF program to a process-specific kernel view. Process kernel page tables are then modified to point to new kernel pages, rather than the original shared pages. Consequently, the eBPF extension is executed only for the process of interest, while other processes continue to use the original, unpatched kernel pages without any performance impact. Our design isolates tracing overhead to the processes being traced, achieving zero untraced overhead per-process tracing.

Providing per-process views of kernel hookpoints also presents new opportunities beyond tracing, such as better support for kernel extension and application acceleration. We identify these opportunities and also discuss challenges around the granularity of kernel views and the complexities of managing per-view kernel state.

2 TRACING WITH EBPF

Tracing is an important and common use case for eBPF programs, depicted in Figure 1. As eBPF programs are loaded dynamically, and are verified safe, they can be deployed on production systems to better understand application performance in production. When an operator wants to trace an application, they must identify a set of hookpoints that are relevant to their application. They will then use

eBPF '24, August 4–8, 2024, Sydney, NSW, Australia

ACM ISBN 979-8-4007-0712-4/24/08.
https://doi.org/10.1145/3672197.3673431

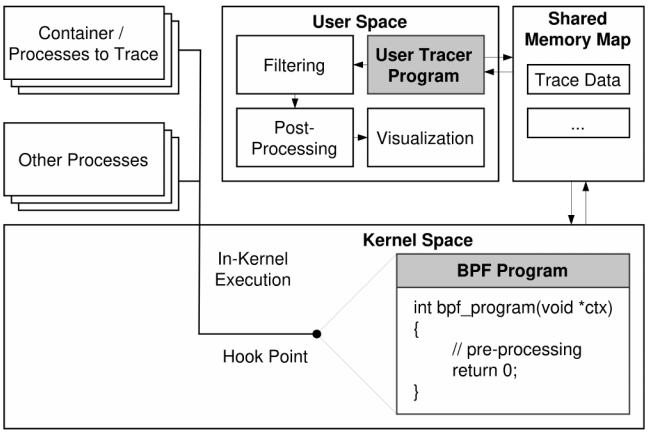

Figure 1: Overview of eBPF based tracing

a tool like bpftrace [4] or will create custom eBPF programs that attach to the identified set of hookpoints. Each program will collect some information when it is triggered, perform a small amount of pre-processing, and finally store the information in a *shared data map*. User space programs can then read data from shared data maps and perform more extensive post-processing and visualization.

To load and attach an eBPF program requires multiple changes to kernel memory, as shown in Figure 2. In addition to allocating memory for the instructions making up the eBPF program, a *tracing hookpoint installer* in the kernel modifies the kernel text to call into a handler, which then calls the eBPF program. For example, the kprobe installer inserts a jump or trapping instruction into the kernel text while the tracepoint installer patches statically inserted no-op instructions. All tracing hookpoints cause changes to be made to kernel text pages when an eBPF program is attached.

Since the kernel is shared, whenever an operator attaches an eBPF program to a hookpoint, it will be invoked regardless of process context, effectively attaching the eBPF program for all processes. The attached extension will execute every time any process triggers the hookpoint. For example, an eBPF program attached to the `sys_enter_read` tracepoint will be executed whenever any process calls the `read` system call. Regardless of what the operator wanted to trace, the system will trace all processes that cause the attached event to be triggered.

2.1 Per-Process Tracing

In many instances, system operators only want to trace an individual application or a container, instead of tracing all running processes on the system. We identify three ways to implement per-process tracing: post-eBPF, in-eBPF, and pre-eBPF filtering.

Post-eBPF Filtering: One approach to per-process tracing is to collect data from all processes and then filter for relevant PIDs in userspace. It is easy to associate PID with collected data if the eBPF program stores the PID with the tracing data. Post-filtering is a natural approach for per-process tracing as it does not require any additional logic in eBPF programs or kernel modifications.

In-eBPF Filtering: Another approach to per-process tracing is to filter by PID inside eBPF programs. In eBPF there is a helper function

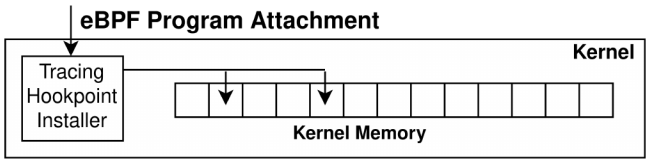

Figure 2: To attach a BPF program, the tracing hookpoint installer modifies kernel memory

`bpf_get_current_pid_tgid()` which allows eBPF programs to determine the PID of the process that triggered them. Operators can statically define a specific PID in their eBPF program that would cause the tracing action to occur. Tools like bpftrace provide builtin functionality to filter out events by PID by generating such a check in the eBPF program. If there is a need for more dynamic PID tracing, PID checking can also be implemented using a shared map containing a list of PIDs to trace. A simple *if* condition within the eBPF program is sufficient for most tracing use cases.

Pre-eBPF Filtering: To avoid the cost of eBPF program startup, a third approach is to implement a check in the kernel before the eBPF program gets executed. For example, one could add a field to the `task_struct` that indicates if a given process should be traced and check the field on each hookpoint before executing eBPF program startup code. Doing so would reduce the overhead on untraced processes as compared to in-eBPF filtering. The pre-eBPF approach is not currently used in practice.

2.2 Overheads of Per-Process Tracing

As eBPF programs are executed whenever any process triggers an attached event, the above strategies produce overhead on untraced processes. Post-eBPF incurs the full cost of tracing, while pre-eBPF and in-eBPF filtering reduce the overhead by bailing out of the eBPF program execution flow early. We ran two experiments to investigate the cost of the above approaches to per-process tracing. The first experiment is a measure of the overhead per system call for each approach, while the second experiment measures the impact of tracing overhead on memcached performance.

2.2.1 Untraced Process System Call Overhead. We performed an experiment to measure system call overhead on untraced process due to per-process tracing approaches. We use a baseline with no eBPF program attached on an unmodified Linux 6.8 kernel to show the best possible performance of the system call. We also create a modified Linux kernel to implement a pre-eBPF PID check. We then implement an eBPF program with a single `bpf_printk()` helper call to represent a tracing load. On the baseline kernel, the overhead for the tracing program is representative of the post-eBPF filtering cost, since post-eBPF filtering is done offline. We use the same tracing program on our modified kernel to show the cost of pre-eBPF filtering. Additionally, we create a modified version of the tracing program that includes a PID lookup and single *if* condition to implement in-eBPF per-process tracing.

We measure the time for two system calls: `read` and `sendmsg`, by calling the function 10,000 times and then dividing by 10,000 to estimate the time each individual system call required. We then repeat this for 10 runs, and compute the average.

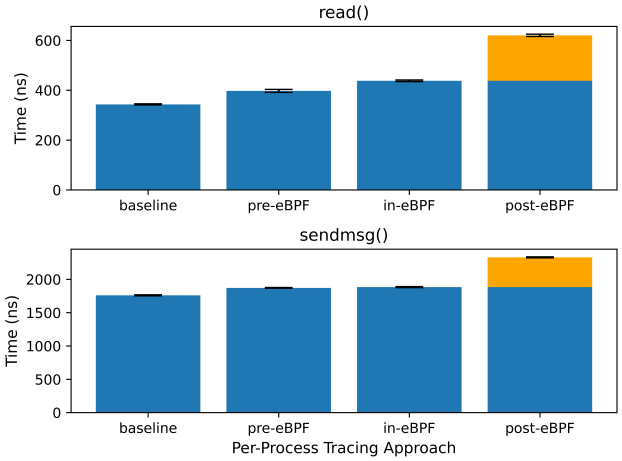

Figure 3: Untraced-overhead due to per-process tracing approaches on `read()` and `sendmsg()` systemcalls. The orange bar represents the cost of the tracing program, which we control. Error bars represent one standard deviation

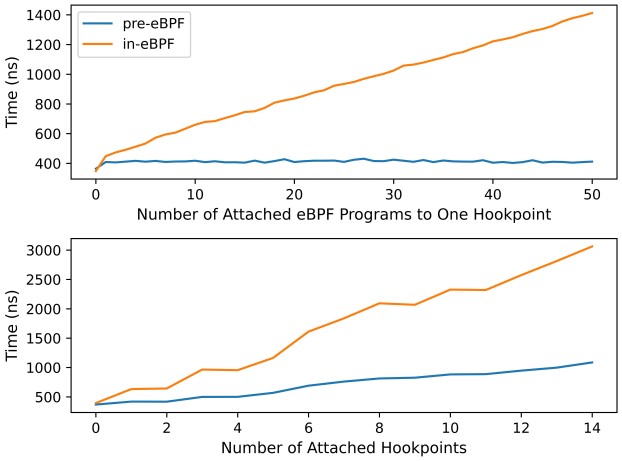

Figure 4: Effects of multiple attachment on untraced overhead

Figure 3 shows the overhead costs on untraced programs. As expected, the most expensive approach was post-eBPF filtering, where the entire tracing program is executed. In-eBPF filtering saves time by avoiding executing the full eBPF program. Pre-eBPF filtering saves even more time by avoiding all BPF program setup and instructions. On our test setup, we find that the most efficient per-process approach, pre-eBPF, slows down the `read` and `sendmsg` system call by 54 and 112 nanoseconds respectively, which correspond to 15% and 6% slowdown. All other approaches are more expensive.

We additionally measure the impact of attaching multiple eBPF programs to a single hookpoint, and attaching individual programs to multiple hookpoints. We use in-eBPF and pre-eBPF test setups as described above. We attach up to 50 tracing programs to the `sys_enter_read` tracepoint to see the impacts of attaching multiple eBPF programs to the same hookpoint. We also attach kprobe eBPF programs to 14 kernel functions that the `read` system call calls, to determine the impact of attaching eBPF programs to multiple hookpoints. Figure 4 shows the results of our experiment. We see that in-eBPF untraced overhead scales both with the number of attached programs to one hookpoint, and with the number of hookpoints attached to. For pre-eBPF, the untraced overhead for attaching to a single hookpoint is constant, while the untraced overhead scales with the number of attached hookpoints.

From the above experiment, we see that untraced overhead for pre-eBPF and in-eBPF scales with the number of attached hookpoints. As eBPF deployments become more complicated, we expect the number of attached hookpoints to increase, which would result in greater untraced overhead.

2.2.2 Untraced Application Performance Degradation. To determine how overheads on system calls due to per-process eBPF programs translate into application performance loss, we ran a macrobenchmark test. We used two dual-core VMs to benchmark a memcached instance. One VM acts as a client and uses memaslap to generate load for the server. Memaslap is configured to run on two threads, with 1000 concurrent connections. It will then execute 6000000 `get` commands and 10240000 `set` commands to measure the throughput of the memcached server. Each test is run five times and the throughputs are averaged.

To simulate tracing an application that sends data over the network, we attach tracing programs to the `sys_enter_read` and `sys_enter_sendmsg` tracepoints. We have two versions of each tracing program. The first simply calls the `bpf_printk()` helper function to print a message, while the second implements a static PID check with the `bpf_get_current_pid_tgid()` helper function, and an *if* condition before the `bpf_printk()` function call.

We ran a total of four different tests. The first test runs on an unmodified kernel, with no eBPF programs attached, representing the baseline performance of our test system. The second test attaches the printing tracing programs to the server, representing the post-eBPF filtering approach. The third test attaches the static PID checking tracing programs, representing the in-eBPF approach. The final test uses the same tracing programs as the second test, but running on our modified kernel that checks PID before running eBPF programs, representing the pre-eBPF approach. Both PID checks will always bail out of eBPF execution when triggered by memcached, as it is not the process we are tracing.

Our baseline test was able to handle 573,939 throughput operations per second (TOPS). As expected, post-eBPF filtering caused the greatest performance degradation at 515,618.2 TOPS. In-eBPF managed 558,363.8 TOPS, while pre-eBPF achieved 565,356.2 TOPS. The measured performance degradations align with the measured costs found in the system call overhead experiment. We find that the most efficient per-process approach, pre-eBPF, causes a 1.5% decrease in throughput, while in-eBPF filtering causes a 2.7% decrease. We conclude that the overheads imposed by existing per-process approaches do translate to application performance degradation. As shown in Figure 4, the performance losses will also scale with the number of eBPF programs attached to different hookpoints,

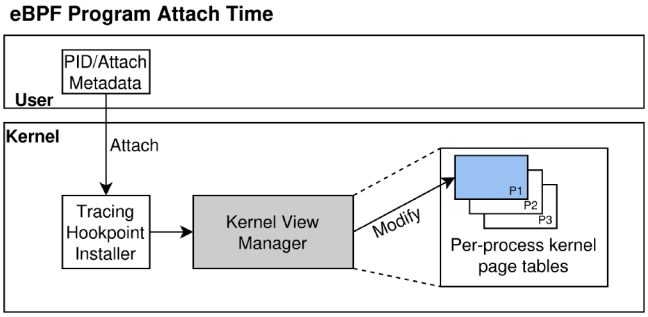

Figure 5: Modified eBPF program attachment path

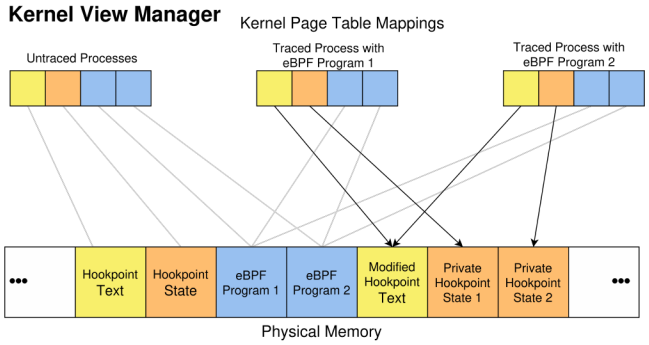

Figure 6: Kernel memory virtualization mechanism overview

as each attached hookpoint has a cost. A more involved tracing scenario would cause even greater performance degradation.

3 PER-PROCESS KERNEL VIEWS FOR PER-PROCESS TRACING

To achieve efficient per-process eBPF tracing, we must reduce untraced overhead, while ensuring that all relevant tracing data is collected. Figure 5 shows an overview of how our system fits into the tracing workflow, while Figure 6 shows the mechanism we use. We identify the two main tracing hookpoints as tracepoints and kprobes. All tracing hookpoints share two facts in common, which we use as our key design insight:

(1) Unattached tracing hookpoints are fast
(2) Attachment involves writing to kernel pages

As described in Section 2, when an eBPF extension is attached to a hookpoint, it calls a tracing hookpoint installer which modifies kernel text and data. In the existing Linux kernel, the tracing hookpoint installer directly modifies the kernel. We propose modifying the tracing hookpoint installer to call into a kernel view manager, as shown in Figure 5. Rather than using the same memory mapping for kernel pages regardless of the process context, the kernel view manager allows for different kernel memory mappings for different processes. The kernel view manager creates copies of to-be-modified kernel pages, and performs the same modifications as the tracing hookpoint installer on the new pages, essentially performing copy-on-write. Afterwards, the kernel view manager modifies the kernel page tables of traced processes to map to the new pages, instead of the existing unmodified pages, as shown in Figure 6. To modify the correct process' page tables, we propose adding a new argument to the eBPF program attachment API, allowing operators to specify the process they want to trace. The end result is a system that allows per-process tracing, with zero untraced overhead. In the rest of this section, we examine the kernel view manager in more detail, discuss open design questions, and raise challenges associated with our design.

3.1 Kernel View Manager

The kernel view manager is the main component of our proposed design, and is responsible for providing per-process hookpoint views by virtualizing tracing hookpoint text and state. When an eBPF extension is attached to a tracing hookpoint, the kernel view manager must make new copies of kernel pages. The first page to copy is the text of the tracing hookpoint. However, the kernel view manager can avoid copying the page if another eBPF program has already been attached. The tracing hookpoint page is modified to call into a tracing hookpoint handler which will contain the same code for any traced process on the same tracing hookpoint.

The kernel view manager must also copy tracing hookpoint state in order to provide a private copy. Tracing hookpoint state includes an *eBPF program array* which contains all the eBPF programs to be executed by the tracing hookpoint handler. Executing eBPF programs associated with other processes not only makes tracing slower, but could impact the correctness of tracing by executing eBPF programs intended for one process, on another. Copying tracing hookpoint state also allows for the optimization for tracing hookpoint text described above.

After the necessary pages have been copied, the kernel view manager must update the kernel page tables for each traced process. To facilitate finding the correct process, eBPF attachment includes an argument to statically add a PID to be traced, similar to how PID is specified in in-eBPF tracing. If PIDs need to be dynamically added, a system call can be used to instruct the kernel view manager to update the page tables of the desired process.

3.2 Open Design Questions

In this subsection, we discuss open design questions raised by our approach that need to be addressed.

Tracing Hookpoint State: The nature of tracing hookpoint state is not clear. We know that at the minimum it includes the eBPF program array, but it may also include other important pieces of data. Additionally, it is not clear that all tracing hookpoints share a common set of necessary tracing hookpoint state.

Runtime Kernel View Manager: There may need to be a runtime component of the kernel view manager. Our current proposed design performs most work during attachment time, avoiding as much runtime cost as possible. However, having different kernel pages tables per process may lead to issues when a CPU currently running a traced process services an interrupt. If the interrupt uses any pages that have been modified, kernel state may become inconsistent.

Fork Semantics: It is unclear how `fork()` should function under our proposed system. Operators interested in tracing a process may

also be interested in tracing child processes. On the other hand, forked processes may be performing entirely different work, and would not make sense to be traced alongside their parent process.

Implications for Other Subsystems: Modifying kernel page tables may also cause problems for other Linux kernel subsystems that interact with kernel text pages, such as SELINUX and the kernel module loader. The changes we make are small and self-contained, so it should be possible to maintain the functionality of any impacted subsystems. There may be more difficult challenges that arise in other subsystems that must be addressed.

Tracing Program Life Cycle: Having per-process views of hookpoints raises the question of what happens when an eBPF program is detached. As hookpoint text is shared, the kernel view manager must ensure that other processes do not have eBPF programs attached to the same hookpoint before it can free the page. It may be inefficient to free the modified hookpoint text pages. The kernel view manager must also know when to free private hookpoint state pages.

Userspace Component: It is unclear how much of a userspace component is needed to facilitate our proposed system. Some modifications need to be made in order to associate eBPF programs with processes, but there may be greater orchestration needs. There may be interactions between maps, or limitations of hookpoints that must be addressed.

Wasted Memory: Depending on the size of the pages that hookpoints are located on, and the number of extensions that are attached, a significant amount of memory could be used. If hookpoints are contained on huge pages, each copy would require 1 GB [7] of data to be duplicated. We must consider the page size and memory layout when designing our system to avoid unnecessary wasted space. We speculate that careful memory layout and the use of 4KB pages may help.

Overhead of Multiple Kernel Views: Switching between kernel memory mappings when switching between processes may increase pressure on TLBs and cause a reduction in performance. However, due to kernel page table isolation, all processes must load a new virtual mapping of the kernel into their address space whenever they perform a mode switch. We may be able to take advantage of the existing page table switch to hide the cost of our page table modifications.

4 BEYOND TRACING

In addition to tracing, eBPF programs have served to extend kernel functionalities by replacing kernel policy and mechanism or accelerating applications. Examples of using eBPF to replace kernel policy and mechanism include attachment to Linux Security Module hookpoints to implement mandatory access control [8], implementation of custom TCP congestion control algorithms [27], implementation of new schedulers [29], and implementation of system call filtering [28]. Per-process hookpoint execution facilitates the efficient implementation of per-process policies, enabling the enforcement of kernel policies tailored exclusively to individual processes without any additional overhead on others.

eBPF-based approaches to application acceleration typically involve increased kernel-side processing to minimize context switches and bypass certain parts of the kernel [52]. Two existing application accelerators, BMC [34] and Electrode [60] bypass the kernel networking stack in application-specific ways. XRP [59] accelerates the storage stack with eBPF but necessitates application modification to realize its benefits. With per-process hookpoints, we could modify the system behavior to match the application, allowing for transparent acceleration of applications and custom kernel interfaces.

In the rest of this section, we discuss two challenges in extending our design for zero-untraced-overhead per-process eBPF tracing to more general extension use cases.

Per-Flow Hookpoint Execution: Our proposed system operates at the granularity of per-process hookpoint execution, but it could be extended to other granularities, such as per-network-flow hookpoint execution. For example, BMC and Electrode operate on network hookpoints in the kernel to accelerate applications. When the attached hookpoints are triggered, the target PID for each packet is not known, which prevents our system from executing eBPF programs on process-specific packets. Instead, we can expand our system to operate on a per-flow basis. Using a multi-queue NIC, we can dedicate CPU cores to handle certain packet flows. For each CPU that accelerates one application, we can change the kernel mappings to present a view of the kernel with the appropriate eBPF programs attached.

Increased Kernel State Management Complexity: Allowing for multiple views of kernel hookpoints at different granularities greatly increases the complexity of managing kernel state. There are three main sources of new state. First, eBPF programs may store private state into multiple maps that are read from and written to by multiple other eBPF programs or userspace programs. In our initial design, eBPF programs and maps are still globally visible, but the execution of eBPF programs is on a per-process basis. Other eBPF programs and userspace programs can make changes to eBPF maps regardless of what process is associated with each eBPF program. The relationship between eBPF programs, maps, processes, flows, etc. introduces state management complexity. Second, the need to manage kernel views in the face of different granularities, such as per-flow views introduces another source of state complexity. For example, the runtime kernel view manager and program lifecycle raised in Section 3 may need to keep track of which flow and process are running on which CPU in order to correctly switch the views. Third, managing eBPF program views of kernel state may introduce significant state management complexity. Tracing programs are relatively simple in that they do not generally modify kernel state. If a more general eBPF extension modifies kernel state in a specific way, or makes assumptions about kernel state, inconsistencies may arise. We want eBPF programs to have their own internal view of kernel state, as they are operating on a per-process view, so they must have their own copy. The extent and complexity of managing kernel state copies may depend on the extension and is an open question.

5 RELATED WORKS

This section reviews the literature on tracing, methods for mitigating the overhead caused by dynamic tracing, and mechanisms for providing multiple kernel views.

5.1 Mitigating Tracing Overhead

Tracing, the process of recording system activity, is commonly achieved through static (i.e., compile-time) or dynamic (i.e., runtime) code instrumentation. Static instrumentation entails injecting monitoring code directly into the source code before compilation, a method employed by tools such as GNU gprof [35]. Conversely, dynamic instrumentation [21, 50], utilized by tools like DTrace [22], ftrace [6], perf [9], and SystemTap [11], dynamically installs customizable instrumentation code based on runtime parameters and system conditions. eBPF has been extensively used for profiling [3, 10, 23, 51], and dynamic tracing across a range of use cases, including Android [1, 12, 44], distributed systems [16, 25, 32, 45, 53, 57], and HPC [56, 58].

Several tracing overhead minimization research endeavors have been carried out in the past. Gebai et al. [33] develop microbenchmarks that unveil insights into the tracers' internals and show the cause of each tracer's overhead. Nagy et al. [49] eliminate unwanted tracing data for their fuzzing. Thomas et al. [54] investigate different instrumentation schemes and propose two new schemes based on bitvectors that reduce the overhead for sampling-based execution monitoring. Mohror et al. [48] thoroughly investigate several different tracing overheads, including trace instrumentation, periodic writing of trace files to disks, differing trace buffer sizes, system changes, and increasing number of processors in the target application. In-advance filtering, such as in-driver filtering [30], processes data before it reaches the tracing programs, ensuring that they handle only relevant data and efficiently utilize resources.

More general approaches to accelerate eBPF programs also can reduce tracing overhead. KFuse [42] merges loaded eBPF programs to improve execution performance. Merlin [47] provides LLVM passes that rewrite eBPF bytecode for efficiency, enhancing execution speed without failing kernel verification.

5.2 Multiple Kernel Views

Several prior works investigate the use of multiple kernels on a single system. Multikernels [17] propose running different kernels on each core, with each kernel and core combination exclusively accessing its own memory. Face-Change [36] and MultiK[40] provide a per-process view of the same kernel in order to debloat the kernel and reduce its attack surface [41, 46]. The per-process view of the kernel explored in these works is appealing for managing per-process tracing and extension.

6 CONCLUSION

Per-process eBPF tracing enables the tracing of applications without adversely affecting other running processes. We propose a system utilizing a kernel view manager to provide each process with its own view of kernel hookpoints, thereby eliminating eBPF tracing overhead on processes that are untraced. Additionally, the system has the potential for expansion, allowing for greater specialization within the kernel and creating new opportunities for application acceleration.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their insightful comments. This work is supported in part by NSF grant CNS-2236966.

ETHICAL CONCERNS

This work raises no ethical concerns.

REFERENCES

[1] 2024. android-profiler. https://developer.android.com/studio/profile/cpu-profiler. (May 2024).
[2] 2024. aquasecurity. https://github.com/aquasecurity/tracee. (May 2024).
[3] 2024. bcc. https://github.com/iovisor/bcc. (May 2024).
[4] 2024. bpftrace. https://github.com/bpftrace/bpftrace. (May 2024).
[5] 2024. Cilium. (May 2024). https://cilium.io
[6] 2024. ftrace. https://www.kernel.org/doc/html/v4.17/trace/ftrace.html. (May 2024).
[7] 2024. HugeTLB Pages. https://docs.kernel.org/admin-guide/mm/hugetlbpage.html. (May 2024).
[8] 2024. LSM BPF Programs. https://docs.kernel.org/bpf/prog_lsm.html. (May 2024).
[9] 2024. perf. https://perf.wiki.kernel.org/index.php. (May 2024).
[10] 2024. sysdig. https://github.com/draios/sysdig. (May 2024).
[11] 2024. systemtap. https://sourceware.org/systemtap/. (May 2024).
[12] 2024. systrace. https://developer.android.com/topic/performance/tracing. (May 2024).
[13] 2024. Tetragon. https://github.com/cilium/tetragon. (May 2024).
[14] Maximilian Bachl, Joachim Fabini, and Tanja Zseby. 2022. A flow-based IDS using Machine Learning in eBPF. (2022). arXiv:cs.CR/2102.09980
[15] Sabur Baidya, Yan Chen, and Marco Levorato. 2018. eBPF-based content and computation-aware communication for real-time edge computing. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. 865–870. https://doi.org/10.1109/INFCOMW.2018.8407006
[16] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. 2004. Using Magpie for Request Extraction and Workload Modelling. In *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*. USENIX Association, San Francisco, CA. https://www.usenix.org/conference/osdi-04/using-magpie-request-extraction-and-workload-modelling
[17] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 29–44. https://doi.org/10.1145/1629575.1629579
[18] Andrew Begel, Steven McCanne, and Susan L. Graham. 1999. BPF+: exploiting global data-flow optimization in a generalized packet filter architecture. *SIGCOMM Comput. Commun. Rev.* 29, 4 (aug 1999), 123–134. https://doi.org/10.1145/316194.316214
[19] Matteo Bertrone, Sebastiano Miano, Fulvio Risso, and Massimo Tumolo. 2018. Accelerating Linux Security with eBPF iptables. *SIGCOMM '18: Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*, 108–110. https://doi.org/10.1145/3234200.3234228
[20] Ashish Bijlani and Umakishore Ramachandran. 2019. Extension Framework for File Systems in User space. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 121–134. https://www.usenix.org/conference/atc19/presentation/bijlani
[21] Derek Bruening, Qin Zhao, and Saman Amarasinghe. 2012. Transparent dynamic instrumentation. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE '12)*. Association for Computing Machinery, New York, NY, USA, 133–144. https://doi.org/10.1145/2151024.2151043
[22] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. 2004. Dynamic Instrumentation of Production Systems. In *2004 USENIX Annual Technical Conference (USENIX ATC 04)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/2004-usenix-annual-technical-conference/dynamic-instrumentation-production-systems
[23] Cyril Renaud Cassagnes, Lucian Trestioreanu, Clément Joly, and Radu State. 2020. The rise of eBPF for non-intrusive performance monitoring. *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium* (2020), 1–7. https://api.semanticscholar.org/CorpusID:219591293
[24] Young Eun Choe, Jun-Sik Shin, Seunghyung Lee, and Jongwon Kim. 2020. *eBPF/XDP Based Network Traffic Visualization and DoS Mitigation for Intelligent Service Protection*. 458–468. https://doi.org/10.1007/978-3-030-39746-3_47
[25] MIchael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F. Wenisch. 2014. The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 217–231. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/chow
[26] Jonathan Corbet. 2019. KRSI — the other BPF security module. (2019). https://lwn.net/Articles/808048/
[27] Jonathan Corbet. 2020. Kernel operations structures in eBPF. https://lwn.net/Articles/811631/. (February 2020).

[28] Jonathan Corbet. 2021. eBPF seccomp() filters. https://lwn.net/Articles/857228/. (May 2021).
[29] Jonathan Corbet. 2023. The extensible scheduler class. https://lwn.net/Articles/922405/. (February 2023).
[30] Luca Deri. 2007. High-Speed Dynamic Packet Filtering. *Journal of Network and Systems Management* 15, 3 (Sept. 2007), 401–415. https://doi.org/10.1007/s10922-007-9070-0
[31] Luca Deri, Samuele Sabella, and Simone Mainardi. 2019. Combining System Visibility and Security Using eBPF. In *Italian Conference on Cybersecurity*. https://api.semanticscholar.org/CorpusID:59616648
[32] Úlfar Erlingsson, Marcus Peinado, Simon Peter, Mihai Budiu, and Gloria Mainar-Ruiz. 2012. Fay: Extensible Distributed Tracing from Kernels to Clusters. *ACM Trans. Comput. Syst.* 30, 4, Article 13 (nov 2012), 35 pages. https://doi.org/10.1145/2382553.2382555
[33] Mohamad Gebai and Michel R. Dagenais. 2018. Survey and Analysis of Kernel and Userspace Tracers on Linux: Design, Implementation, and Overhead. *ACM Comput. Surv.* 51, 2, Article 26 (mar 2018), 33 pages. https://doi.org/10.1145/3158644
[34] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. 2021. BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 487–501. https://www.usenix.org/conference/nsdi21/presentation/ghigoff
[35] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. 2004. gprof: a call graph execution profiler. *SIGPLAN Not.* 39, 4 (apr 2004), 49–57. https://doi.org/10.1145/989393.989401
[36] Zhongshu Gu, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. 2014. FACE-CHANGE: Application-Driven Dynamic Kernel View Switching in a Virtual Machine. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 491–502. https://doi.org/10.1109/DSN.2014.52
[37] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. 2018. 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 (CoNEXT '18)*. Association for Computing Machinery, New York, NY, USA, 54–66. https://doi.org/10.1145/3281411.3281443
[38] Jibum Hong, Seyeon Jeong, Jae-Hyoung Yoo, and James Won-Ki Hong. 2018. Design and Implementation of eBPF-based Virtual TAP for Inter-VM Traffic Monitoring. In *2018 14th International Conference on Network and Service Management (CNSM)*. 402–407.
[39] Kornilios Kourtis, Animesh Trivedi, and Nikolas Ioannou. 2020. Safe and Efficient Remote Application Code Execution on Disaggregated NVM Storage with eBPF. (2020). arXiv:cs.DC/2002.11528
[40] Hsuan-Chi Kuo, Akshith Gunasekaran, Yeongjin Jang, Sibin Mohan, Rakesh B. Bobba, David Lie, and Jesse Walker. 2019. MultiK: A Framework for Orchestrating Multiple Specialized Kernels. *CoRR* abs/1903.06889 (2019). arXiv:1903.06889 http://arxiv.org/abs/1903.06889
[41] Hsuan-Chi Kuo, Jianyan Chen, Sibin Mohan, and Tianyin Xu. 2020. Set the Configuration for the Heart of the OS: On the Practicality of Operating System Kernel Debloating. *Proc. ACM Meas. Anal. Comput. Syst.* 4, 1, Article 03 (may 2020), 27 pages. https://doi.org/10.1145/3379469
[42] Hsuan-Chi Kuo, Kai-Hsun Chen, Yicheng Lu, Dan Williams, Sibin Mohan, and Tianyin Xu. 2022. Verified programs can party: optimizing kernel extensions via post-verification merging. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 283–299. https://doi.org/10.1145/3492321.3519562
[43] Chang Liu, Zhengong Cai, Bingshen Wang, Zhimin Tang, and Jiaxu Liu. 2020. A protocol-independent container network observability analysis system based on eBPF. 697–702. https://doi.org/10.1109/ICPADS51040.2020.00099
[44] Yu Luo, Kirk Rodrigues, Cuiqin Li, Feng Zhang, Lijin Jiang, Bing Xia, David Lion, and Ding Yuan. 2022. Hubble: Performance Debugging with In-Production, Just-In-Time Method Tracing on Android. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 787–803. https://www.usenix.org/conference/osdi22/presentation/luo
[45] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2016. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO. https://www.usenix.org/conference/atc16/technical-sessions/presentation/mace
[46] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: library operating systems for the cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. Association for Computing Machinery, New York, NY, USA, 461–472. https://doi.org/10.1145/2451116.2451167
[47] Jinsong Mao, Hailun Ding, Juan Zhai, and Shiqing Ma. 2024. Merlin: Multi-tier Optimization of eBPF Code for Performance and Compactness. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 639–653. https://doi.org/10.1145/3620666.3651387
[48] Kathryn Mohror and Karen L. Karavanic. 2007. A study of tracing overhead on a high-performance linux cluster. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '07)*. Association for Computing Machinery, New York, NY, USA, 158–159. https://doi.org/10.1145/1229428.1229465
[49] Stefan Nagy and Matthew Hicks. 2019. Full-Speed Fuzzing: Reducing Fuzzing Overhead through Coverage-Guided Tracing. In *2019 IEEE Symposium on Security and Privacy (SP)*. 787–802. https://doi.org/10.1109/SP.2019.00069
[50] V. Prasad, William Cohen, F. Eigler, M. Hunt, J. Keniston, and B. Chen. 2005. Locating system problems using dynamic instrumentation. (01 2005).
[51] R. Sekar, H. Kimm, and R. Aich. 2024. eAUDIT: A Fast, Scalable and Deployable Audit Data Collection System. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 90–90. https://doi.org/10.1109/SP54263.2024.00087
[52] Farbod Shahinfar, Sebastiano Miano, Giuseppe Siracusano, Roberto Bifulco, Aurojit Panda, and Gianni Antichi. 2023. Automatic Kernel Offload Using BPF. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems (HOTOS '23)*. Association for Computing Machinery, New York, NY, USA, 143–149. https://doi.org/10.1145/3593856.3595888
[53] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Technical Report. Google, Inc. https://research.google.com/archive/papers/dapper-2010-1.pdf
[54] Johnson J. Thomas, Sebastian Fischmeister, and Deepak Kumar. 2011. Lowering overhead in sampling-based execution monitoring and tracing. In *Proceedings of the 2011 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES '11)*. Association for Computing Machinery, New York, NY, USA, 101–110. https://doi.org/10.1145/1967677.1967692
[55] Mathieu Xhonneux, Fabien Duchene, and Olivier Bonaventure. 2018. Leveraging eBPF for programmable network functions with IPv6 segment routing. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '18)*. ACM. https://doi.org/10.1145/3281411.3281426
[56] Stephen Yang, Seo Jin Park, and John Ousterhout. 2018. NanoLog: A Nanosecond Scale Logging System. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 335–350. https://www.usenix.org/conference/atc18/presentation/yang-stephen
[57] Lei Zhang, Zhiqiang Xie, Vaastav Anand, Ymir Vigfusson, and Jonathan Mace. 2023. The Benefit of Hindsight: Tracing Edge-Cases in Distributed Systems. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 321–339. https://www.usenix.org/conference/nsdi23/presentation/zhang-lei
[58] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. 2017. Log20: Fully Automated Optimal Placement of Log Printing Statements under Specified Overhead Threshold. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 565–581. https://doi.org/10.1145/3132747.3132778
[59] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. 2022. XRP: In-Kernel Storage Functions with eBPF. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 375–393. https://www.usenix.org/conference/osdi22/presentation/zhong
[60] Yang Zhou, Zezhou Wang, Sowmya Dharanipragada, and Minlan Yu. 2023. Electrode: Accelerating Distributed Protocols with eBPF. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 1391–1407. https://www.usenix.org/conference/nsdi23/presentation/zhou