*A Seminar Report on*

Eliminating eBPF Tracing Overhead

on Untraced Processes

*Submitted in partial fulfillment for the award of the degree of*

BACHELOR OF TECHNOLOGY

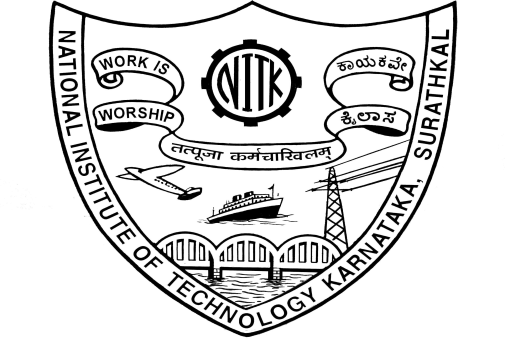
*in*

*ARTIFICIAL INTELLIGENCE*

*by*

Sambhav Singh 231AI033

*IV Sem B.Tech (AI)*



Department of Information Technology

National Institute of Technology Karnataka, Surathkal.

*April 2025*

**CERTIFICATE**

This is to certify that the seminar entitled “**Eliminating eBPF Tracing Overhead on Untraced Processes**” has been presented by ***Sambhav Singh (2310209)*** student of **IV semester B.Tech (AI)**, Department of Information Technology, National Institute of Technology Karnataka, Surathkal, during the even semester of the academic year **2024 – 2025.** It is submitted to the Department in partial fulfillment of the requirements for the award of the degree of Bachelor of Technology in Information Technology.

Place:

Date: (Signature the Examiner1)

Place:

Date: (Signature of the Examiner2)

Place:

Date: (Signature of the Coordinator)

**DECLARATION BY THE STUDENT**

I hereby declare that the Seminar (IT290) entitled Eliminating eBPF Tracing Overhead on Untraced Processes was carried out by me during the even semester of the academic year 2024 – 2025 and submitted to the department of IT, in partial fulfillment of the requirements for the award of the Degree of Bachelor of Technology in the department of Information Technology, is a bonafide report of the work carried out by me. The material contained in this seminar report has not been submitted to any University or Institution for the award of any degree.

Place: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Date: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(Name and Signature of the Student)

**TABLE OF CONTENTS**

|  |  |  |
| --- | --- | --- |
| Sl. No | Chapter | Page Number |
|  | Abstract | 7 |
| 1 | Introduction | 9 |
| 2 | Literature Survey | 13 |
| 3 | Methodology | 16 |
| 4 | Simulation Results and Discussion | 22 |
| 5 | Conclusion and future work | 28 |
|  | Reference | 29 |

**LIST OF FIGURES**

|  |  |  |
| --- | --- | --- |
| Fig [1] | Overview of ebpf based tracing | 10 |
| Fig [2] | Ebpf program attachment | 11 |
| Fig [3] | Modified eBPF attachment path | 20 |
| Fig[4] | Kernel memory virtualization | 21 |
| Fig[5] | Untraced-overhead due to per-process tracing ap- proaches on read() and sendmsg() systemcalls. | 23 |
| Fig[6] | Effects of multiple attachment on untraced over-  head | 25 |
|  |  |  |
|  |  |  |

ABSTRACT

**LIST OF TABLES**

|  |  |  |
| --- | --- | --- |
| Table 1. | Comparison of filtering methods | 11 |
| Table 2. | Test reults of system call overhead | 27 |
| Table  3. | Test result of application overhead | 23 |

**ABSTRACT**

Extended Berkeley Packet Filter (eBPF) is a modern Linux kernel technology that allows user-defined programs to be executed at various points in the kernel, commonly used for observability tasks like tracing system calls or network events. These programs are typically attached to kernel hookpoints and triggered whenever relevant kernel activity occurs. However, once attached, an eBPF program applies globally to all processes interacting with that hookpoint, even if the user is only interested in monitoring a specific set of processes, such as those belonging to a particular container. This broad scope results in unnecessary overhead for untraced processes—processes that are not the intended target of tracing—because they still interact with the same instrumented kernel paths.

To reduce this unintended overhead, filtering strategies are often introduced to make tracing more selective. One method is post-eBPF filtering, where data from all processes is collected and filtering is done afterward in user space, discarding events from processes that are not of interest. This approach is easy to implement but still requires executing the eBPF program for all events, leading to wasted computation. Another common method is in-eBPF filtering, where the filtering logic is embedded inside the eBPF program itself, using kernel-provided helper functions like bpf\_get\_current\_pid\_tgid() to identify the calling process. This allows the program to discard events early but still incurs execution overhead every time the hookpoint is triggered. A more proactive approach is pre-eBPF filtering, which attempts to prevent irrelevant processes from even invoking the eBPF code, usually by checking process identifiers at the kernel level before the eBPF program is executed. While this reduces some overhead, it still introduces extra logic into kernel paths and does not eliminate the eBPF system’s presence from the execution flow of untraced processes.

Our analysis shows that despite their differences, all three filtering approaches introduce measurable performance overhead even for untraced processes. This is because the presence of eBPF instrumentation itself—even if conditionally bypassed—still affects the kernel control flow and performance characteristics. To fully eliminate this overhead for processes that are not being traced, we propose a new system design based on creating per-process views of the kernel. Our approach modifies the way kernel code pages are mapped into each process’s virtual memory space. Specifically, when an eBPF program is attached to a kernel hookpoint, a kernel view manager initiates a copy-on-write mechanism to duplicate the kernel code pages affected by the eBPF hook. These modified pages are then made visible only to the traced processes. Meanwhile, untraced processes continue to execute with the original, unmodified kernel code, effectively bypassing all eBPF instrumentation.

This mechanism ensures that eBPF tracing overhead is confined strictly to the intended target processes, while the rest of the system operates as if no eBPF program were present. By introducing virtual memory-level isolation for kernel instrumentation, our design provides a much finer-grained and efficient mechanism for selective tracing, paving the way for safer and more scalable deployment of eBPF in multi-tenant or containerized environments.

**CHAPTER 1**

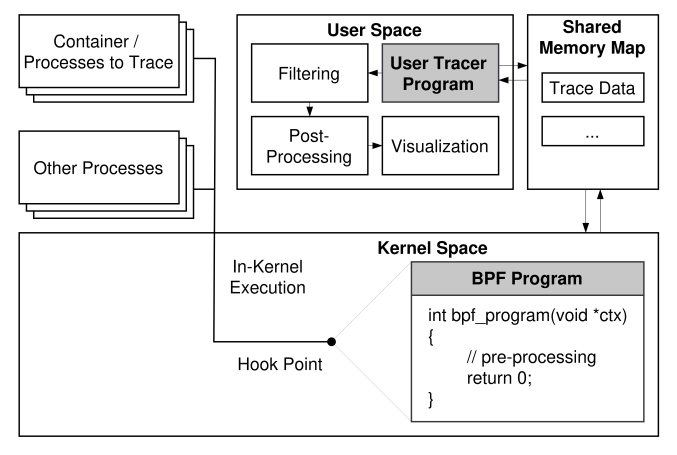
**INTRODUCTION**

Introduction

eBPF (Extended Berkeley Packet Filter) is a powerful feature in the Linux kernel that allows developers to run small, safe programs inside the kernel at runtime. These programs are verified by the kernel to ensure safety and correctness, meaning they can’t crash the system or perform dangerous operations like invalid memory access. Once verified, the eBPF code is just-in-time (JIT) compiled and executed very efficiently. This makes eBPF ideal for real-time monitoring and customization of kernel behavior without requiring changes to kernel source code or restarts.

eBPF is used for a wide range of tasks, including:

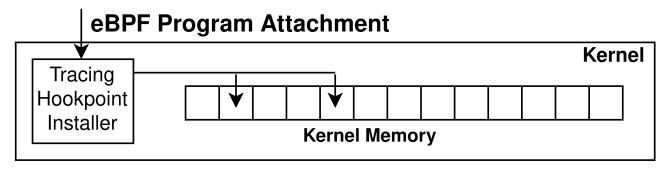
* Packet filtering: Filtering incoming/outgoing network packets for purposes like firewalls (e.g., XDP, Cilium).
* Profiling: Observing how applications and the kernel use CPU time, memory, or system calls (e.g., using tools like perf, bcc, and bpftrace).
* Application acceleration: Offloading parts of application logic into the kernel to avoid context switches, improving performance.
* Security enforcement: Monitoring and restricting behaviors at runtime (e.g., syscall policies).
* System tracing and observability: Tracking what’s happening in the system over time, useful for debugging and performance analysis.

Figure 1 : overview of eBPF based tracing

The way eBPF works is by attaching programs to hookpoints in the kernel. A hookpoint is a specific place in the kernel code (like sys\_open() or a TCP handler) where the kernel allows eBPF code to be executed when that point is hit. For example, if a user attaches an eBPF program to a system call hookpoint, the code will run every time that system call is made.

This event-driven model is especially useful for system tracing, where developers want to monitor what applications and the operating system are doing in real time. eBPF makes this possible with low overhead and without needing special kernel builds.

However, current eBPF implementations apply globally. When a tracing program is attached to a hookpoint, that program executes for every process that triggers the event—even if the user only wants to monitor a few specific processes, such as those belonging to a container. This limitation creates unnecessary overhead for unrelated processes that have nothing to do with the tracing task.

Figure 2 : attaching eBPF program

To make tracing more selective, users rely on different types of filtering techniques. These methods try to skip tracing work for processes that are not of interest.

| **Method** | **Where Filtering Happens** | **How It Works** | **Drawback** |
| --- | --- | --- | --- |
| Post-eBPF Filtering | In user space, after eBPF collects data | The eBPF program logs all events, and the user discards irrelevant ones later | eBPF still runs for every process; wasted computation |
| In-eBPF Filtering | Inside the eBPF program itself | Uses helper functions like bpf\_get\_current\_pid\_tgid() to check current process | Partial filtering, but eBPF still triggers on all processes |
| Pre-eBPF Filtering | Before eBPF is called (in kernel) | Modifies kernel to check PID before invoking eBPF | Requires changes to kernel hookpoint paths; still overhead |

Table 1 : comparison of filtering methods

bpf\_get\_current\_pid\_tgid() is a standard helper in eBPF that returns the current process ID (PID) and thread group ID (TGID). It helps eBPF code decide if the current event belongs to a process of interest.

* User space refers to normal application code (outside the kernel), where post-processing of eBPF data often happens.
* Kernel space is where the eBPF program itself runs, allowing deep insight into system internals.
* Context switch is when the CPU switches between user and kernel modes—something eBPF helps avoid when optimizing performance.

Despite the cleverness of these methods, none of them fully eliminate the presence of eBPF hooks in the execution path of untraced processes. Even if the eBPF program takes no action for an untraced process, the kernel still needs to run the code, check conditions, and possibly pass context data—leading to what we call untraced overhead.

To solve this problem, we propose a new technique: per-process kernel hookpoint views. The idea is to let each process have its own view of the kernel’s code at the hookpoint level. So instead of all processes sharing the same instrumented kernel page, we allow selective patching of only the pages used by the processes we want to trace.

Here’s how it works:

* When a user attaches an eBPF program to a hookpoint, we use a component called a kernel view manager.
* This manager finds the kernel code page containing the hookpoint and makes a copy-on-write (COW) copy.
* Copy-on-write means the page isn’t duplicated until a change needs to be made, saving memory.
* The traced process’s page table is updated to point to the new (modified) version of the kernel page.
* All other processes continue using the original page, which has no eBPF code attached.

This setup ensures the eBPF code only runs when needed.

**CHAPTER 2**

**LITERATURE REVIEW**

**2.1 Eliminating Tracing Overhead for Untraced Processes**

The first paper, "Eliminating eBPF Tracing Overhead on Untraced Processes" (Milo Craun et al., 2024), introduces a novel architectural solution to a key limitation in today’s eBPF tracing: its system-wide effect. While eBPF tracing is intended to observe selected processes, existing filtering techniques (post-eBPF, in-eBPF, and pre-eBPF) still impose measurable overhead on untraced processes because the kernel hookpoints are shared system-wide.

The authors propose a kernel view isolation mechanism, which modifies the virtual memory layout of the kernel code on a per-process basis. When a tracing program is attached, a copy-on-write (COW) of the kernel page is made, and only the traced processes see the modified code. Other processes continue using the original, uninstrumented version of the kernel. This is implemented via a kernel view manager, which tracks and applies memory mapping changes securely and efficiently.

This approach achieves zero untraced overhead, a significant breakthrough for production tracing workloads in multi-tenant environments. The authors evaluate their design on standard benchmarks and demonstrate significant reductions in overhead compared to traditional eBPF filtering. Their work also lays the groundwork for per-process customization of kernel behavior, opening new possibilities for kernel extension, acceleration, and isolation beyond tracing.

**2.2 eBPF-mm: Userspace-Guided Memory Management**

In “eBPF-mm: Userspace-guided Memory Management in Linux with eBPF”(Konstantinos Mores & Stratos Psomadakis et al., 2024),the authors shift the focus of eBPF-based optimization toward the memory management subsystem. Traditional Linux support for huge pages (e.g., 2MiB, 1GiB on x86; up to 32MiB on RISC-V/ARM) aims to reduce TLB misses and improve performance. However, static policies like Transparent Huge Pages (THP) often result in suboptimal or harmful behavior, particularly in fragmented or low-memory systems.

To address this, eBPF-mm introduces user-guided huge page decision-making, leveraging eBPF to inject dynamic logic into the page fault path. A new hookpoint is added in the page fault handler. When a fault occurs, an eBPF program checks whether the memory region is profiled, estimates the cost and benefit of using a huge page (based on fragmentation, zeroing cost, and access frequency), and selects the most appropriate page size.

Profiles are created using DAMON, a Linux framework for monitoring dynamic memory access patterns. These profiles encode expectations of access frequency and are used by eBPF to make real-time promotion decisions. Early results show that eBPF-mm achieves performance competitive with THP while using fewer huge pages, by focusing only on hot regions of memory.

The framework is lightweight, maintains compatibility with existing kernel behavior, and imposes no overhead on non-participating processes. Future directions include integrating with asynchronous promotions (khugepaged) and extending to policies like memory tiering and reclamation, demonstrating how eBPF can provide fine-grained memory control beyond tracing.

**2.3 Understanding Performance of eBPF Maps**

The third work, “Understanding Performance of eBPF Maps” (Chang Liu & Byungchul Tak , 2023), provides an in-depth benchmarking and profiling study of the performance characteristics of various eBPF map types. Maps are critical to eBPF programs as they act as data stores and communication channels between user space and kernel space. However, the performance impact of different map types, sizes, operations, and hardware configurations is poorly documented.

This study implements a custom benchmarking tool (bpf\_bench) to measure lookup and update latencies for various map types including arrays, hash maps, per-CPU variants, ring/perf buffers, stacks, and queues. The benchmarking accounts for factors such as cache hotness, value size, and concurrency. For example, it reveals that while per-CPU hash maps reduce contention, they suffer from significant write overhead when inserting new keys with large values due to memory amplification.

Key insights include:

* Cold cache access to maps can be an order of magnitude slower than hot cache access.
* Memory footprint and access pattern (e.g., random vs sequential) greatly affect performance.
* Ring buffers outperform perf buffers, especially when using the reserve+submit method, due to fewer memory copies.
* Volume discount: attaching the same eBPF program to more syscalls amortizes infrastructure costs, reducing per-call overhead.

These findings help developers understand hidden bottlenecks and tune their programs for specific workloads. The work highlights the need for performance-aware usage of eBPF, particularly in high-frequency tracing or real-time applications.

**CHAPTER 3**

**Methodology**

This section presents the eBPF overhead assessment via eBPF tracing (Section 3.1) with the proposed programming framework in detail (Sections 3.1.1–3.1.4) and the eBPF overhead reduction via per process kernel views(section 3.2) with the programming framework(Sections 3.2.1)

**3.1 Tracing with eBPF**

This section describes how eBPF is used for tracing purposes in production systems, and explains the internal steps and components involved in loading and executing eBPF-based tracing programs. It also outlines different strategies to apply tracing only to specific processes, and the setup used to study the overhead that different per-process tracing methods introduce.

**3.1.1 Setting Up Tracing with eBPF**

eBPF (extended Berkeley Packet Filter) is commonly used for tracing applications running on production systems. A typical tracing setup starts with an operator deciding they want to monitor the behavior or performance of a running application. To do this, the operator first identifies hookpoints in the kernel—these are locations in kernel code where an eBPF program can be attached to observe specific events, such as system calls or function entries.

After identifying the right hookpoints, the operator either writes a custom eBPF program or uses a tool like bpftrace to generate one. These eBPF programs collect data when triggered, do some basic processing, and store the results in shared maps that can be accessed by user-space programs for further processing or visualization.

Attaching an eBPF program requires multiple changes to the kernel’s memory. First, the kernel allocates memory to store the bytecode of the eBPF program. Then, depending on the type of hookpoint (e.g., kprobe or tracepoint), the kernel modifies its own code (kernel text) to redirect execution to a handler that runs the attached eBPF program. For example:

* kprobes insert jump or trap instructions at runtime.
* tracepoints replace no-op instructions with a call to the handler.

Because the Linux kernel is shared by all processes, once an eBPF program is attached to a hookpoint, it will run for every process that triggers that hookpoint—not just the one the operator was interested in. So, if a tracing program is attached to sys\_enter\_read, it will execute for all processes that call read(), even if the operator only wanted to trace one specific application.

**3.1.2 Per-Process Tracing Strategies**

In real-world usage, system administrators or developers often want to trace only one process or container rather than everything on the system. To make this possible, there are three major ways to filter out unnecessary tracing: post-eBPF, in-eBPF, and pre-eBPF filtering.

* Post-eBPF Filtering  
  In this approach, the eBPF program runs for every process, but the data is filtered in user space after it's been collected. The eBPF program saves the PID (Process ID) along with the data it gathers, and user-space tools filter it later. This method is simple and doesn't require extra logic in the kernel or eBPF code, but it still executes the full eBPF program for all processes.
* In-eBPF Filtering  
  This method moves the filtering logic inside the eBPF program itself. eBPF has a helper function called bpf\_get\_current\_pid\_tgid() that lets the program find out which process triggered it. The developer can write an if statement to only collect data for specific PIDs. For more dynamic cases, a shared map can hold a list of allowed PIDs, which the eBPF program checks against before proceeding. Tools like bpftrace can automatically generate such filtering logic.
* Pre-eBPF Filtering  
  This more advanced method avoids running the eBPF program entirely for processes that don’t match the tracing target. One idea is to store a flag in each process’s task\_struct to indicate whether it should be traced. The kernel would then check this flag at the hookpoint before running the eBPF program. This method is the most efficient because it skips all the overhead of eBPF startup, but it requires kernel modification and is not yet widely used.

**3.1.3 Measuring Overhead from Tracing**

To understand how these per-process tracing methods affect overall system performance, we designed two types of experiments. The first type measures the extra time added to system calls when tracing is used, especially when the process being traced is not the one triggering the hookpoint. The second type evaluates how these added delays affect the real-world performance of applications like memcached.

To carry out the first experiment, we start by setting up a baseline system running the standard Linux 6.8 kernel with no eBPF program attached. Then, we use a modified version of the kernel that adds pre-eBPF filtering capabilities by checking a PID flag in task\_struct. Next, we create a simple eBPF program that uses bpf\_printk() to simulate basic tracing activity. We prepare three versions:

1. A standard version with no filtering (represents post-eBPF),
2. A version with a static PID check in the program (in-eBPF),
3. And a version for the modified kernel that avoids running the program unless the process should be traced (pre-eBPF).

We focus on two common system calls: read() and sendmsg(). We call each function 10,000 times in a loop and measure the total time taken, dividing it by 10,000 to estimate the average time per call. We repeat the entire process 10 times to average the results and reduce measurement noise.

To further examine the scalability of overheads, we set up additional test conditions. These include:

* Attaching multiple eBPF programs (up to 50) to a single hookpoint,
* Attaching eBPF programs to multiple hookpoints that a single system call might go through (e.g., kprobes on internal functions called by read()).

These setups allow us to see how performance changes as more tracing points and programs are added, even if the running process is not being traced.

**3.1.4 Application-Level Benchmark Setup**

To study the effect of tracing overhead on a real application, we deploy a memcached instance on a dual-core virtual machine. We use another dual-core VM running the memaslap tool to send requests to the server. The client is configured to use two threads and 1000 concurrent connections. Each test run involves 6 million get commands and over 10 million set commands.

We repeat the tests under four different tracing configurations:

1. Baseline: No eBPF programs attached (used as a reference point),
2. Post-eBPF: Tracing programs attached with no filtering,
3. In-eBPF: Tracing programs with PID checks inside the eBPF code,
4. Pre-eBPF: Kernel-level PID filtering used to prevent unnecessary tracing.

For the in-eBPF and pre-eBPF setups, the PID check is designed to always reject memcached's PID so that it simulates being an untraced process. This helps measure how much unnecessary overhead is still incurred, even when the process isn't supposed to be traced.

We run each configuration five times and calculate the average throughput to get a stable measurement of performance.

**3.2 Per-Process Kernel Views for Per-Process Tracing**

To trace only selected processes without slowing down others, the system must separate tracing logic per process. The goal is to collect tracing data just for chosen processes, and do so efficiently. To explain the idea, the authors refer to Figures 5 and 6, which show how their solution fits into the larger tracing process and how it works internally.

There are two main types of points in the kernel where tracing programs can be attached: tracepoints and kprobes. These hookpoints have two key traits:

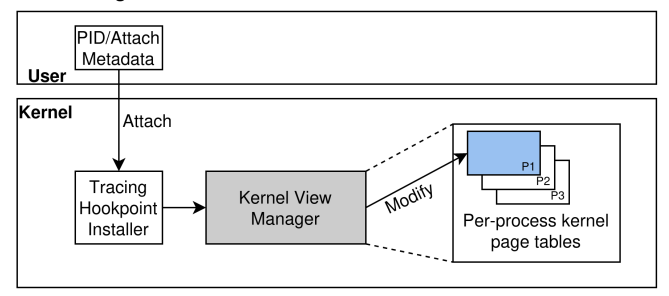
1. If nothing is attached to them, they run very fast.
2. If something is attached, the kernel’s memory gets changed.

In the Linux kernel today, attaching an eBPF program changes the actual kernel memory. The authors suggest changing this approach. Instead of directly modifying shared kernel memory, they introduce a kernel view manager. This component allows each process to have its own version of certain kernel pages.

Here’s how it works: when an eBPF program is attached, the kernel view manager copies the kernel page that’s about to be modified. It then applies the change to the copy, not the original. This is similar to how copy-on-write works. After that, only the traced process is made to use the modified page, while all other processes continue using the original unmodified page.

To apply this change to the right process, they suggest adding a new option to the eBPF attachment API. This option will let users say exactly which process they want to trace. The result is a tracing system that adds no extra cost to untraced processes.

The rest of the section looks more closely at how the kernel view manager works, and then talks about several design questions and possible challenges.

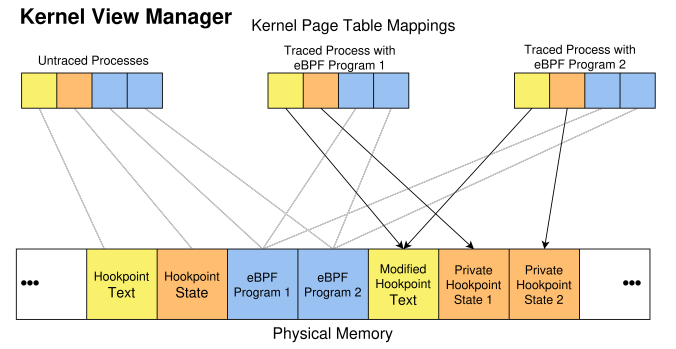
Figure 3 : Modified eBPF program attachment path

**3.2.1 Kernel View Manager**

The kernel view manager is the main part of this idea. It handles creating separate hookpoints for each process by copying both the tracing code and any related data (called tracing hookpoint state). When an eBPF program is attached, the manager starts by copying the kernel code that the program needs to change. If another eBPF program is already attached to the same code, this copy may not be needed again. In that case, all traced processes can use the same handler.

Next, the manager copies the tracing state. This includes the list of eBPF programs that will run when the hookpoint is triggered. Each traced process needs its own copy of this list, so one process’s programs don’t run in another. This separation also helps the system reuse the same hookpoint code safely.

Once all the copies are made, the view manager updates the page tables of the process being traced, so it sees the modified kernel pages. The eBPF API is extended to accept a process ID (PID) during attachment. If more processes need to be traced later, a system call can update their page tables as needed.

Figure 4 : Kernel memory virtualization

**CHAPTER 4**

**Results**

## **4.Results**

This section presents the evaluation of untraced overheads across different per-process eBPF tracing methods. The experiments include microbenchmarks on system call latency, macrobenchmarks measuring application-level throughput, and multi-program scalability tests. The goal of these evaluations is to understand how post-eBPF, in-eBPF, and pre-eBPF filtering approaches impact processes that are not intended to be traced.

**4.1. Microbenchmark: System Call Overhead on Untraced Processes**

We first evaluate the tracing overhead on untraced processes using two standard system calls: read() and sendmsg(). The purpose of this experiment is to understand how each per-process tracing approach affects system call latency when invoked by processes not being traced.

To measure this:

* Each system call was executed 10,000 times, and the total duration was divided by 10,000 to estimate the average time per call.
* The experiment was repeated across 10 runs, and the mean value was reported.

The test was conducted under the following configurations:

* Baseline: No eBPF program attached, using the unmodified Linux 6.8 kernel.
* Post-eBPF: An eBPF program with a bpf\_printk() call is attached, and data is filtered in userspace.
* In-eBPF: The same program includes a static PID check using bpf\_get\_current\_pid\_tgid() and an if condition.
* Pre-eBPF: A kernel-modified setup with a PID check that avoids calling the eBPF program if the process is not targeted.

Results of this experiment are visualized in Figure 5, which displays the system call latency incurred under each method for read() and sendmsg().

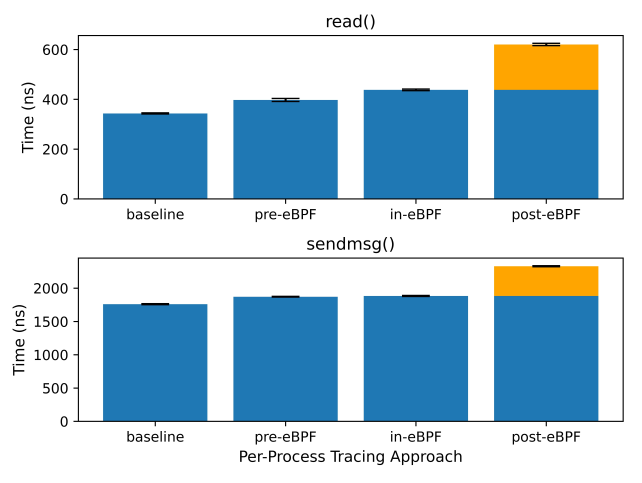


Figure 5: Untraced-overhead due to per-process tracing ap- proaches on read() and sendmsg() systemcalls.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Method | read() Overhead (ns) | read() Slowdown (%) | sendmsg() Overhead (ns) | sendmsg() Slowdown (%) |
| Baseline | 0 | 0% | 0 | 0% |
| Post-eBPF | Not explicitly stated | – | Not explicitly stated | – |
| In-eBPF | Higher than Pre-eBPF | >15% | Higher than Pre-eBPF | >6% |
| Pre-eBPF | 54 | 15% | 112 | 6% |
|  |  |  |  |  |

Table 2 . Test results of system call overhead

Notably:

* Pre-eBPF resulted in the lowest overhead for untraced processes, with an increase of only 54 ns for read() and 112 ns for sendmsg().
* This corresponds to a 15% slowdown for read() and 6% for sendmsg() compared to the baseline.
* In-eBPF incurs more overhead than pre-eBPF due to partial eBPF execution before PID filtering occurs.
* Post-eBPF introduces the highest overhead, since it executes the full tracing logic for all processes regardless of their relevance.

**4.1. Microbenchmark: System Call Overhead on Untraced Processes**

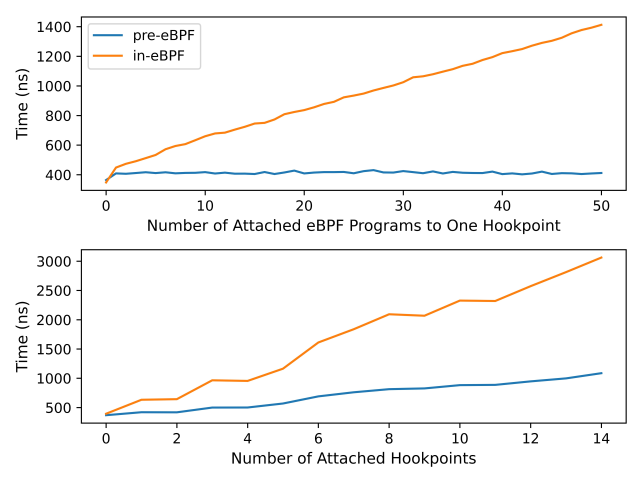
The next experiment investigates how untraced overheads scale as more eBPF programs and hookpoints are involved.

Two configurations were tested:

* Multi-Program to Single Hookpoint: Up to 50 eBPF programs were attached to the sys\_enter\_read tracepoint.
* Multi-Hookpoint Attachment: eBPF programs were also attached to 14 internal kernel functions that are invoked by the read() syscall using kprobes.

Both in-eBPF and pre-eBPF approaches were used in these scenarios to analyze scaling behavior.

Findings from this experiment are summarized in Figure 6, which shows the trend of increasing untraced overhead with the number of programs and hookpoints.

Figure 6 : Effects of multiple attachment on untraced over-

head

Key observations:

* For in-eBPF, overhead increases:
  + With the number of eBPF programs attached to a single hookpoint.
  + And with the number of hookpoints that a single system call triggers.
* For pre-eBPF, overhead remains constant for a single hookpoint, even with multiple programs attached.
* However, pre-eBPF overhead does scale with the number of hookpoints, as each one requires a separate PID check.

These results suggest that while both approaches avoid the full execution cost of post-eBPF, only pre-eBPF can limit per-hookpoint costs under multiple program attachment scenarios.

**4.3. Macrobenchmark: Impact on Application Performance**

To assess the real-world implications of tracing overhead, we ran a macrobenchmark using memcached and memaslap.

The setup consists of:

* Two dual-core VMs.
* One VM runs a memcached server.
* The other acts as a client, using memaslap configured with 2 threads and 1000 concurrent connections.

The workload executed:

* 6,000,000 get commands
* 10,240,000 set commands

Each test configuration was executed five times, and the average throughput in throughput operations per second (TOPS) was recorded.

We tested four configurations:

1. Baseline: No eBPF programs attached.
2. Post-eBPF: Tracing programs with bpf\_printk() attached.
3. In-eBPF: Programs include static PID checks inside the eBPF code.
4. Pre-eBPF: Same programs as (2), but running on a kernel modified to check the PID before executing any eBPF logic.

In both (3) and (4), the tracing programs are designed to bail out when triggered by memcached, ensuring that the application is treated as untraced in all non-baseline configurations.

Throughput results:

* Baseline: 573,939 TOPS
* Post-eBPF: 515,618.2 TOPS
* In-eBPF: 558,363.8 TOPS
* Pre-eBPF: 565,356.2 TOPS

This experiment demonstrates that:

* Post-eBPF causes the highest performance degradation (10.2% drop from baseline).
* In-eBPF reduces the degradation to 2.7%.
* Pre-eBPF has the least impact, with only a 1.5% drop from baseline.

This performance loss directly correlates with the system call latency results discussed earlier.

Figure 4 includes the visualization of performance impact across these configurations.  
Results are in above figure.

|  |  |  |
| --- | --- | --- |
| Method | Throughput (TOPS) | Throughput Drop from Baseline (%) |
| Baseline | 573,939 | 0% |
| Post-eBPF | 515,618.2 | 10.2% |
| In-eBPF | 558,363.8 | 2.7% |
| Pre-eBPF | 565,356.2 | 1.5% |

Table 3 : Test results of application overhead

**CHAPTER 5**

**CONCLUSION AND FUTURE WORKS**

Our experiments show that all current methods for per-process eBPF tracing—whether filtering after, inside, or before the eBPF program—still slow down processes that are not being traced. This slowdown gets worse as more hookpoints and programs are added, making it hard to scale tracing in real systems.

The main idea we learn is that system-wide hookpoint attachment is the core problem. To fix this, we explore per-process kernel views, where each process can have its own version of the kernel’s hookpoints. This lets us attach eBPF programs only for the specific processes we care about. Other processes see unmodified kernel code and stay unaffected.

Per-process kernel views completely remove overhead for untraced processes and allow more flexible kernel behavior per process. This idea can also help with process-specific kernel features like custom security rules or faster I/O paths. However, it brings new challenges—like handling forked processes, managing memory use, and keeping track of different views. Still, this approach offers a strong path forward for making eBPF tracing more efficient and scalable.

Future Work

There are several open challenges to fully realizing per-process kernel views. We need a better understanding of what state each tracing hookpoint maintains and whether it's consistent across different types. A runtime component of the kernel view manager may be necessary to handle edge cases like interrupts or process forks, which could cause inconsistencies. It's also unclear how to manage memory usage efficiently, especially when hookpoints reside on large pages. Other concerns include deciding how forked child processes should inherit tracing views, ensuring compatibility with existing subsystems like SELinux, managing the life cycle of detached eBPF programs, and handling userspace integration. Solving these issues is key to making the system reliable, scalable, and practical for real-world use.

**REFERENCES**

1. “Eliminating eBPF Tracing Overhead on Untraced Processes"  
   Authors: Craun, Hussain, et al. (2024)  
   Proposes a system that allows for zero-untraced-overhead per-process eBPF tracing by modifying kernel virtual memory mappings to present per-process kernelviews.  
   [DOI: 10.1145/3672197.3673431](https://dl.acm.org/doi/10.1145/3672197.3673431)
2. “The eBPF Runtime in the Linux Kernel"  
   Authors: Gbadamosi, Leonardi, Pulls, et al. (2024)  
   Provides a comprehensive description of the design and implementation of the eBPF runtime in the Linux kernel, highlighting its safety and efficiency.  
   [arXiv:2410.00026](https://arxiv.org/abs/2410.00026)
3. "eBPF-mm: Userspace-guided Memory Management in Linux with eBPF"  
   Authors: Mores, Psomadakis, Goumas (2024)  
   Introduces a mechanism that provides the kernel with hints regarding the benefit of promoting a page to a specific size using eBPF.  
   [arXiv:2409.11220](https://arxiv.org/abs/2409.11220)
4. "Understanding Performance of eBPF Maps"  
   Authors: [Not specified] (2024)  
   Analyzes the performance characteristics of eBPF maps, which are crucial for data storage and communication in eBPF programs.  
   [DOI: 10.1145/3672197.3673430](https://dl.acm.org/doi/10.1145/3672197.3673430)
5. "DINT: Fast In-Kernel Distributed Transactions with eBPF"  
   Authors: Zhou, Xiang, Kiley, et al. (2024)  
   Designs a new distributed transaction system that offloads common operations to tc and XDP using eBPF.  
   [USENIX NSDI 2024](https://www.usenix.org/system/files/nsdi24-zhou-yang.pdf)
6. "bpftime: Userspace eBPF Runtime for Uprobe, Syscall and Kernel-User Interactions"  
   Authors: Zheng, Yu, Yang, et al. (2023)  
   Introduces a userspace eBPF runtime that leverages binary rewriting to implement uprobe and syscall hook capabilities.  
   [arXiv:2311.07923](https://arxiv.org/abs/2311.07923)
7. "eBPF-based Working Set Size Estimation in Memory Management"  
   Authors: Lian, Li, Chen, et al. (2023)  
   Proposes a framework to efficiently estimate working set size using eBPF, reducing overhead compared to traditional methods.  
   [arXiv:2303.05919](https://arxiv.org/abs/2303.05919)