International Journal of Networking and Computing – www.ijnc.org, ISSN 2185-2847
Volume 15, Number 2, pages 65–84, July 2025

An eBPF-based packet capture system with embedded application metadata for network forensics

Masaya Okabe
Graduate School of Engineering
Tohoku Institute of Technology
Sendai, Miyagi, 982-8577, Japan
m232801@st.tohtech.ac.jp

Hiroshi Tsunoda
Department of Information and Communication Engineering
Faculty of Engineering
Tohoku Institute of Technology
Sendai, Miyagi, 982-8577, Japan
tsuno@tohtech.ac.jp

Received: February 15, 2025
Revised: May 5, 2025
Accepted: May 28, 2025
Communicated by Takashi Yokota

Abstract

In network forensics, identifying applications involved in packet transmission and reception is crucial for reconstructing the chain of events in security incidents. However, since captured packets do not contain information about specific applications, investigators must rely on other information like log data for identification, which decreases the efficiency and accuracy of the forensic process. This paper proposes a new system that uses an extended Berkeley Packet Filter (eBPF) to embed application metadata directly into the packet capture files. To demonstrate the feasibility of this concept, we implemented a prototype of the proposed system. The system associates each packet with the corresponding application name, process ID, and user ID, storing this metadata alongside packet data in PCAPNG format, enabling analysis with existing tools such as Wireshark. An experimental evaluation comparing the system's performance to a conventional packet capture tool revealed challenges, such as packet loss due to buffer overwriting and increased resource consumption. In particular, the initial Python-based implementation recorded a packet loss rate of 55.61%, which was improved to 7.60% with the enhanced Go-based implementation. However, the proposed system increases CPU utilization by up to 22 percentage points, thus it needs further effort for optimization. Despite remaining performance challenges, the proposed approach has the potential to reduce analysis time and improve accuracy in network forensics by eliminating reliance on log data.

Keywords: network forensics, network security, incident response, packet capture, extended Berkeley Packet Filter (eBPF)

[0]This paper is an extended version of the presented paper in the CANDAR Workshop 2024.

1 Introduction

Network forensics, a branch of digital forensics, is the process of extracting legal evidence from network communications and devices related to an incident and its significance is growing with the increasing sophistication of cyberattacks [1, 2].

Forensics investigation relies on diverse information sources, such as network packets captured in designated binary files and log data in plain text files on the target device. Packet data is a particularly crucial source because it records actual network communications. Investigators use this data to reconstruct a chain of events by analyzing network communication – determining when connections were initiated, who initiated them, which applications and specific services were involved, and which endpoints were accessed.

Packet data and log data are the primary sources for network forensics, and they have different characteristics and limitations. Although packet data contains detailed network communication records, it lacks information about the processes that generate the traffic. Log data complements this by providing application metadata such as application name, process ID (PID), and user ID (UID), which are essential for identifying the specific applications and users involved in network communications, but it may be incomplete or unreliable. Therefore, effective network forensics requires cross-referencing these different data types while preserving their relationships. When this correlation becomes difficult due to incomplete or unreliable log data, investigators often have to infer missing information based on limited evidence in their analysis [3].

This study addresses the challenge of correlating packet data with application information in network forensics. We propose a new packet capture system that embeds application metadata directly into packet data to enhance network forensic efficiency. This approach eliminates the need to cross-reference log data during analysis, enables direct identification of applications, and reduces the need to infer missing information from incomplete evidence. Specifically, we addressed the challenge of maintaining reliable correlation between packets and applications by leveraging the extended Berkeley Packet Filter (eBPF) [4] to associate packet data with applications. The eBPF capability of providing a wide range of information from the kernel space to user space helps us to obtain metadata about applications that transmit monitored packets. The obtained metadata is embedded in the *opt_comment* option of PCAP Next Generation (PCAPNG) capture file format [5] along with packet data, allowing analysis with common tools supporting this format. To demonstrate the feasibility of our approach, we first developed a Python-based prototype system. Through experiments with this prototype, we identified performance-related challenges. A more efficient Go-based implementation provided significant performance improvements. Additionally, we explored directions for further optimization using this enhanced implementation.

The remainder of the paper is organized as follows: Section 2 provides background and related work. It presents an overview of network forensics, outlines its key challenges, and introduces eBPF as a key technology for our proposal. Section 3 outlines the system design for the preservation method and details the prototype implementation. Section 4 evaluates the system's performance through experiments and verifies the effects of implementation and setting improvements. Finally, Section 5 presents the conclusion and future directions.

2 Background and related work

2.1 Network forensics and its key challenges

Network forensics involves capturing and analyzing network traffic to extract forensic evidence from communications and affected devices. It has two primary objectives: (1) detecting security incidents through anomalous traffic patterns and identifying their source via protocol analysis, and (2) obtaining legal evidence by analyzing communications at both the packet level and the application level. To establish forensic validity, it is also essential to determine compromised privileges, attack vectors, and exploited vulnerabilities [6].

Packet data, which are the basic unit of network communication, is a primary source of information in network forensics. By analyzing packets, investigators can determine communication timing,

exploited privileges, accessed resources, and potentially exfiltrated data, thereby reconstructing the sequence of events to reveal the full scope of an incident.

A key characteristic of packets is their dynamics and volatility [6]. Thus, this characteristic necessitates appropriate preservation of volatile data to reconstruct network communications and system states during forensic analysis. To enable reconstruction of communications during an incident afterwards, these packets must be recorded in binary files with chronological information at the bit level.

When properly captured and preserved, packets provide invaluable forensic evidence. Multiple studies [2, 7, 8] emphasize the necessity of complete packet capture. Packets contain not only packet data but also transmitted files themselves, which can be extracted and utilized as evidence using protocol analyzers such as Wireshark [9]. A comprehensive review by S. Khan et al. [10] demonstrates that network forensics can reconstruct communication contents, such as email and FTP traffic, to reveal the nature of attacks. Furthermore, network forensics enables addressing a wide range of network challenges, including identifying attack devices, stopping malware, improving network performance, and monitoring routine traffic. However, these packet-level analyses primarily focus on the network communication aspects of security incidents.

Though packet-level analysis forms the foundation of network forensics, it represents only one layer of the investigative process. To conduct comprehensive forensic investigations, additional analytical approaches are necessary. A key challenge in network forensics is the lack of direct correlation between packet data and the application processes that generate it [11]. The packet payload may contain application-related information; however, payload inspection is often difficult due to encryption and privacy concerns [12]. Additionally, data protection policies and legal constraints must be considered. As a result, packet headers are sometimes the only available source of information for network forensics, though they often lack sufficient detail. Figure 1 illustrates this issue. Packet headers contain network identifiers such as IP addresses, port numbers, and transport layer details (e.g., protocol type), whereas they inherently lack application metadata such as application name, PID, and UID. The application name helps intuitively understand which application was used, the PID enables the identification of specific application instances, and the UID identifies the person who executed the application. This disconnects between a packet and an application process complicates forensic analysis, making it difficult to trace network activity back to specific applications or users.

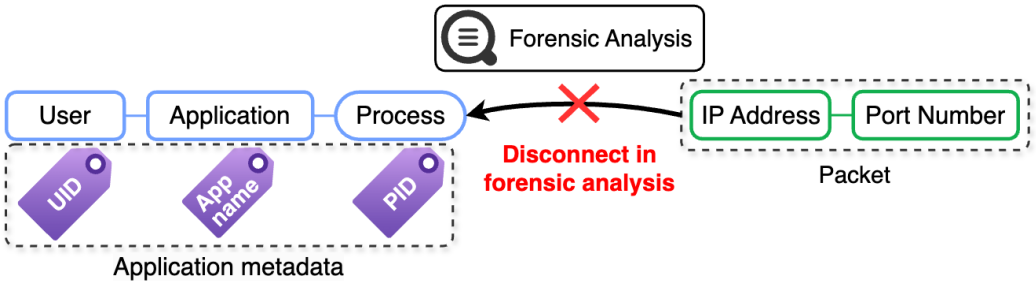

Figure 1: Disconnect in forensic analysis caused by the missing correlation between packet data and application metadata

The correlation between packet data and processes deteriorates over time for several reasons, reducing the accuracy of forensic analysis. The content of log data depends on the application-specific logging behavior. Some applications do not generate logs at all, while others produce logs that lack sufficient details for forensic investigation. Log data may also be systematically removed through log rotation or intentionally erased or tampered with by attackers, further complicating analysis. These limitations in log data preservation often force forensic investigators to rely on incomplete circumstantial evidence — such as partial log entries, residual packet data, or indirect indicators of activity — to reconstruct the chain of events, significantly reducing the reliability of their conclusions [3, 11].

Furthermore, current forensic analysis faces significant operational challenges. Investigators must collect and correlate information from multiple sources, including packet captures, system applications. This manual correlation process is not only time-consuming but also prone to errors. In addition, it is heavily dependent on investigator expertise [11]. The lack of efficient correlation mechanisms further compounds these challenges, particularly in large-scale investigations involving massive amounts of network traffic.

To address these challenges, we propose a method to store application metadata in packet capture files. This approach eliminates complex data correlation, reducing analysis time and enhancing analytical accuracy in network forensics. As part of this effort, our proposed system leverages eBPF-based packet capture to directly embed application metadata within packet data, improving forensic efficiency and reliability.

2.2 Packet processing technologies for network forensics

Network forensics relies on various technologies for efficient packet capture and processing. These include hardware-based solutions that offload tasks to specialized devices for enhanced performance, and software-based methods that utilize general-purpose packet monitoring libraries and low-overhead kernel-space processing.

Regarding hardware-based solutions, Field Programmable Gate Array (FPGA) [13] and Data Plane Development Kit (DPDK) [14] are typical examples. FPGAs consist of arrays of programmable logic gates that can be configured to form circuits specialized for packet processing tasks, thereby accelerating performance. DPDK is a user-space library that bypasses the kernel's network stack, allowing direct transfer of received packets from network interface cards (NICs) to user space. It achieves high-speed packet processing by reducing context-switching overhead through polling-based reception rather than traditional interrupt-based methods.

Various studies are being conducted to achieve high-speed packet processing utilizing these technologies. "hXDP," [15] is an FPGA-based framework for high-speed packet processing that enables execution of Linux's eXpress Data Path (XDP) [16] programs written in eBPF on FPGA hardware. While FPGAs provide hardware acceleration specialized for specific processing tasks, their development requires specialized expertise and presents limitations in flexibility [17]. Thus, Salva-Garcia et al. [17] proposed an XDP-based SmartNIC hardware acceleration approach that combines eBPF and XDP to offload network functions to SmartNICs, achieving the enhancement of processing capabilities.

Toke et al. [18] investigated packet processing performance across DPDK, XDP, and conventional Linux kernel mechanisms. Their experimental results demonstrated that DPDK outperforms XDP. However, despite these performance advantages, DPDK necessitates re-implementation of operating system network functionality within user space applications [17]. Additionally, the busy polling approach utilized for packet processing results in fixed 100% CPU utilization [18].

Compared to hardware-based solutions, software-based methods typically offer lower processing performance. However, they surpass hardware-based approaches in terms of flexibility and extensibility, allowing for more customizable functionality. Libpcap [19] is a widely recognized software library for packet capture. It has served as the foundation for packet capture tools such as tcpdump [19] and Wireshark. Libpcap incorporates the Berkeley Packet Filter (BPF) [20], now sometimes referred to as classic BPF (cBPF), which represents a typical software-based approach to packet filtering. The original BPF was introduced in 1993 as an efficient, kernel-level packet filtering mechanism.

Although cBPF provided significant improvements for packet filtering at the time of its introduction, it was constrained by architectural limitations such as a limited instruction set, fixed register size, and lack of interaction with other kernel subsystems [21]. These limitations led to the development of eBPF, which significantly expanded the capabilities of the cBPF design while maintaining backward compatibility.

eBPF is a technology that can safely extend the functionality of the kernel. It was merged into the Linux kernel in 2014 and since then has been used for various purposes such as networking, improving observability, profiling, and security. To utilize this technology, a developer writes an

eBPF program that implements the desired functions to run in the kernel space. The eBPF program is then attached to a selected hook point, which is a specific location in the kernel or a kernel-level event where the program is triggered. There are various ways to attach eBPF programs, such as through XDP, Tracepoints, or Kprobes. Each of these mechanisms provides different types of hook points within the kernel. The developer selects an appropriate hook point based on the event or condition they would like to monitor or intercept. The eBPF program runs when the target event or condition associated with the attached hook point occurs.

In recent years, eBPF technology has been focused in kernel-level system monitoring, performance analysis, and threat detection. Cilium [22] provides a networking solution that significantly improves the efficiency, security, and observability of container networks in Kubernetes. This solution detects packets using XDP and forwards them to the appropriate Pod. Packet forwarding to Pod requires traversing the network stack twice, but this process can be bypassed by using eBPF and thus the performance can be improved [23]. D. Soldani et al. state that advanced security forensics can be provided for mobile networks through packet inspection leveraging eBPF's observability capabilities [24]. Existing non-uniform network measurement methods are inefficient in cloud-based designs. To address this issue, they developed Sauron, a high-throughput and low-overhead data collection mechanism using eBPF. L. Zhang et al. proposed an eBPF-based Dynamic Perimeter, a Software Defined Perimeter for data centers [25]. Data centers must process a large number of short-lived streams, such as these generated distributed denial of service attacks, making the Single Packet Authorization process inefficient. By efficiently embedding authentication data into packets and verifying them using XDP before they enter the network stack, connection latency improved by 80% compared to existing solutions. Wüstrich et al. [26] employed eBPF to construct network profiles for characterizing typical application behavior. To associate packets with application processes, they developed an eBPF-based matcher and demonstrated its effectiveness for anomaly detection and service dependency identification.

eBPF has been widely adopted for a variety of purposes, whereas research on its performance issues and solutions has progressed in recent years. Liu et al. [27] conducted performance benchmarking of eBPF maps, a data storage structure accessible by both eBPF and userspace programs. They reported that memory usage and cache conditions significantly impact the overhead of accessing eBPF maps. Craun et al. [28] investigated the overhead in per-process tracing, a common use case for eBPF programs, and found that tracing overhead affects not only targeted but also untraced processes. To address this, they proposed a kernel modification that enables zero overhead for untraced processes.

We adopt a software-based approach using eBPF, which enables efficient packet processing while preserving the correlation between packets and application metadata. Compared to hardware-based solutions such as FPGA and DPDK, eBPF offers a practical balance between flexibility and overhead. Thus, it is suitable for initial proof-of-concept implementation.

Our work shares a core idea with Wüstrich et al. [26] – mapping packets to application processes using eBPF – but differs in its primary objective. Their system focuses on real-time profiling of network applications for anomaly detection and dependency analysis, but our goal is to support post-incident forensic analysis. To achieve this, we propose a mechanism that embeds application metadata directly into packet records, enabling persistent and analyzable traces for later investigation.

The next section describes our system design and how eBPF is used to embed application metadata into captured packets.

3 An eBPF-based packet capture system with embedded application metadata

In this section, we describe the system design and prototype implementation of the proposed system.

3.1 System design

The proposed system is designed to capture network packets along with application metadata, enabling identification of the applications involved in network communications. Figure 2 depicts the design concept of our proposed system. The proposed system captures packets and collects three key application metadata elements: PID, UID, and application name. The packets, PIDs, and UIDs are obtained in kernel space through eBPF, while the application name is obtained in user space. All obtained data are stored into PCAPNG file.

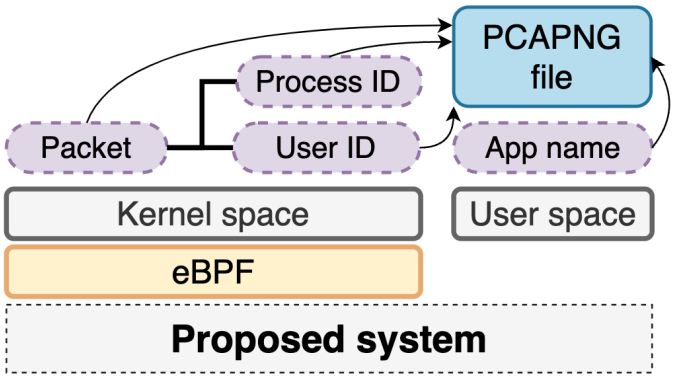

Figure 2: The concept of the proposed system

Figure 3 gives the detailed design of the proposed system. It illustrates the components of the proposed system and their interactions. The system consists of three key modules: a packet handler, an application resolver, and a packet archiver.

The packet handler serves as the central coordinator, interfacing directly with both the application resolver and packet archiver. When packets are captured, the packet handler requests application information from the application resolver, which maintains an up-to-date mapping of PIDs to application names. Once the capture is complete, the packet handler signals the packet archiver to store the accumulated data and metadata.

The functionality of each module and corresponding operation are described below.

Packet handler

This module is the core of the system. It is responsible for capturing packets and retrieving the metadata (corresponding PID and UID) for each packet, using eBPF technology. Additionally, this module includes the front-end program, which provides the user interface and coordinates with the other two modules.

Application resolver

This module manages the mapping between application names and PIDs. It retrieves the list of running processes and the corresponding application name for each process. When the packet handler calls this module with a PID, it identifies the corresponding application name and returns it.

Packet archiver

This module is responsible for storing obtained data in PCAPNG format. Unlike traditional PCAP format, PCAPNG format can embed additional information as packet comments. The use of PCAPNG format ensures that the required data is preserved in a single file, ready for forensic analysis.

The overall operation of this system is as follows: once the user starts the packet handler via the front-end program, an eBPF program is injected into the kernel space. The eBPF program captures sending and receiving packets and then retrieves the corresponding PIDs and UIDs for each packet.

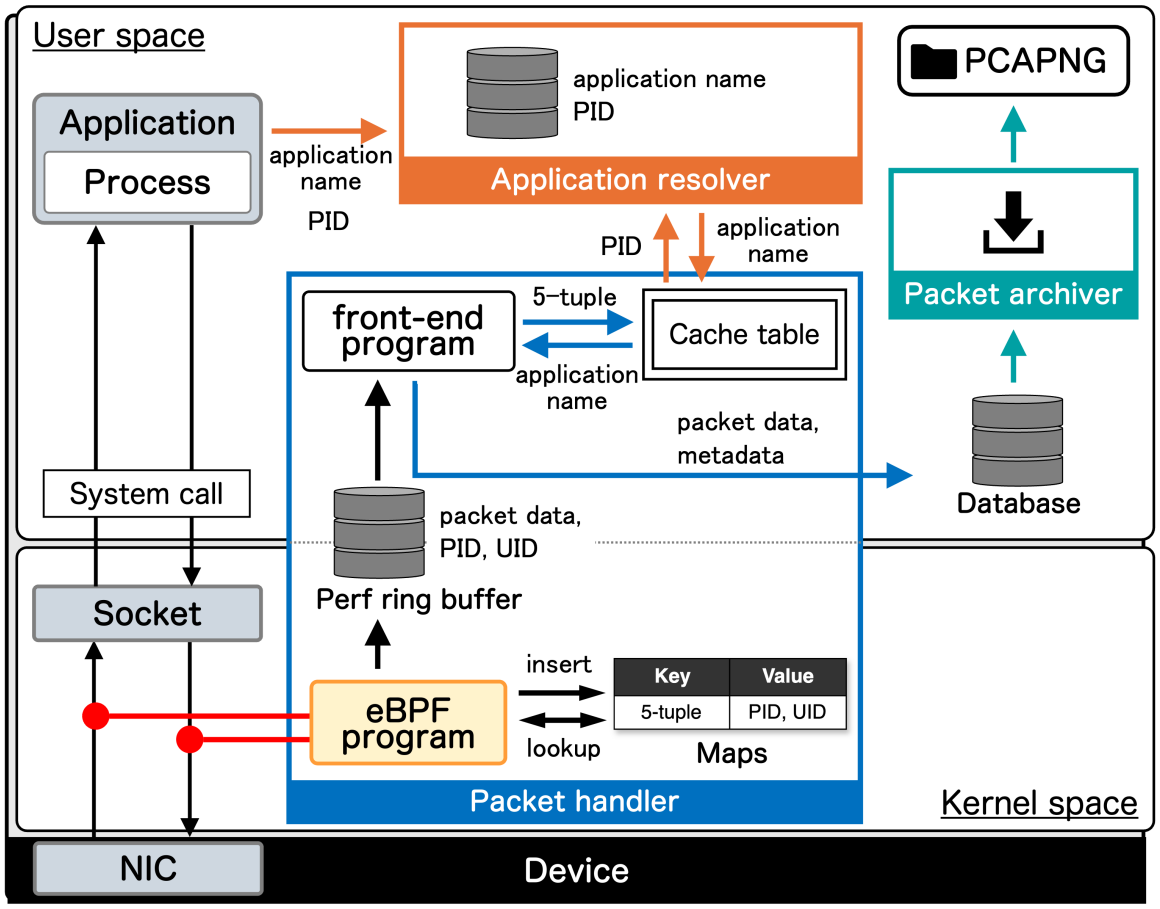

Figure 3: Detailed design of the proposed system

The packets and their metadata are then transferred from kernel space to user space using the perf ring buffer [29]. If the packet handler does not already know the corresponding application name of the captured packet, it calls the application resolver with the PID from the metadata to obtain the application name. The packet handler then caches the obtained application name along with the packet's 5-tuple (source/destination IP address, source/destination port number, and transport protocol) for future reference. During the packet capture, the packet handler accumulates packet data, application name, PID, and UID in the database. Once the packet capture is stopped by the user, the packet archiver reads the accumulated packet data and application metadata from the database and writes it to a file.

3.2 Prototype implementation

The three modules were implemented in Python for initial development, taking advantage of its rich set of eBPF-related tools. The implementation details of each module are explained below.

The eBPF-based functionality of the packet handler was implemented using *bcc* (BPF Compiler Collection) [30], which provides an API for generating and managing eBPF programs from Python scripts. To the best of the author's knowledge, there is no single hook point that can obtain PIDs while also observing both incoming and outgoing packets. Therefore, two hook points were used for observation. The first hook point used is *socket_filter*, which can monitor both incoming and outgoing packets. The second hook point is *__dev_queue_xmit*, provided by Kprobes, which monitors only outgoing packets and can obtain the PID of each packet's sending or receiving process. To associate packet data with the corresponding PIDs, the Maps data structure [31] provided by eBPF was used. The 5-tuple of outgoing packets observed at *__dev_queue_xmit* is used as a key, with the PID and UID inserted as values. Subsequently, the 5-tuple of packets observed by *socket_filter* is used as a key to lookup the corresponding PID and UID in the Maps data structure.

The application resolver functionality was implemented using a Python library called *psutil* [32] to obtain application names. This functionality is used to map application names to individual packets. When the packet handler receives application names from this module, it stores the application metadata along with the packet data in the database table. Furthermore, it maintains a cache table to enable the assignment of application names to packets with the same 5-tuples without going through the module again.

The preservation functionality of the packet archiver was implemented by saving packet data containing bytes, along with application names, PIDs, and UIDs, in the PCAPNG file using the Python library called *python-pcapng* [33]. In the PCAPNG file, each packet is defined by an Enhanced Packet Block (EPB). This block contains Options field, which allows embedding arbitrary UTF-8 strings using the *opt_comment* option. When the user stops capturing packets, the accumulated packet data and metadata in the database are written in bulk to the file.

3.3 Proof of concept

To demonstrate the feasibility of our concept, we show that Wireshark, a widely used protocol analyzer, can correctly process the PCAPNG file generated by our system, and simultaneously display both the packet data and its corresponding metadata. Figure 4 shows how Wireshark displays the contents the PCAPNG file. In the *Comment* column for each packet, PID, UID, and application name are displayed.

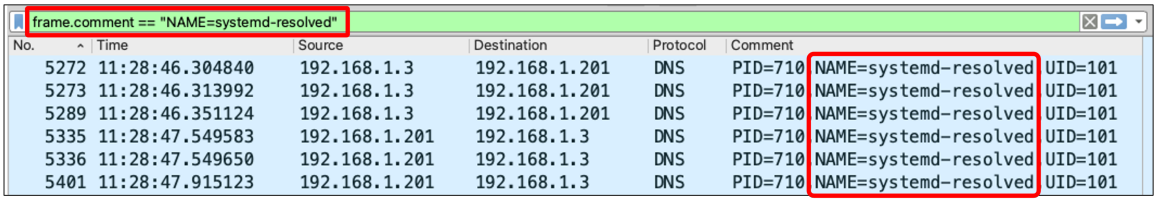

Figure 4: PCAPNG file generated by the packet archiver

It is important to note that Wireshark's display filter is applied to show only the packets associated with the specific application (*systemd-resolved*) in Fig. 4. This demonstrates that applications sending or receiving packets can be identified without relying on log data during the forensic analysis. This aligns with the research objectives by reducing the forensic analysis time and improving accuracy through leveraging reliable evidence obtained via our proposed method, instead of relying on incomplete information.

The proposed system will need more resources compared to conventional packet capture tools. This is because it performs extra processing tasks during capture such as, associating PIDs with captured packets and identifying and storing application names. Therefore, we conducted experiments to assess the current limitations of the proposed system focusing on the capture performance and amount of resource consumption. The next section presents our experimental results and discusses the potential areas for improvement based on the results.

4 Performance evaluation

4.1 Experimental environment

We deployed the prototype implementation on a Linux virtual machine (VM) and evaluated its packet capture performance. The VM runs on VMware ESXi. Table 1 summarizes the physical CPU, assigned vCPUs, memory, storage, and other relevant specifications.

Table 1: Host Specifications

Item	**Specification**
CPU	Intel Xeon Gold 6240R CPU @ 2.40GHz
Cores	8 vCPUs
Memory	8 GB
Storage	100 GB (NVMe)
OS	Ubuntu Desktop 24.04.1 LTS
Kernel version	6.8.0-51-generic
Link speed	1 Gbps

We evaluated the packet capture performance of the proposed system using tcpdump as a baseline for comparison, with both systems running on the same host. Both systems simultaneously captured identical network traffic, and their packet capture counts were compared to assess performance. A capture filter that monitors only TCP and UDP packets was applied to both. The perf ring buffer size in the proposed method for kernel-to-user space data transfer was set to 1MB, the default value in tcpdump.

4.2 Performance evaluation of the prototype implementation

This experiment captures web traffic to evaluate how the capture process performs during normal browsing. Various types of web sites were accessed using Google Chrome and Microsoft Edge, controlled via the Selenium browser automation framework [34]. The accessed web sites include a source code hosting site, a search engine, a video-sharing platform, a news site, a site with JavaScript-based animation, and other static sites.

Figure 5 presents a time-series graph depicting the number of packets captured by both tools. The x-axis represents the elapsed time since the capture started, while the y-axis shows the number of packets captured. The packet volume of this traffic varies due to different content being delivered for each access.

As shown in the figure, the proposed system achieved comparable performance to tcpdump for packet rates up to 1,000 packets per second (pps). However, at higher packet rates, some packets

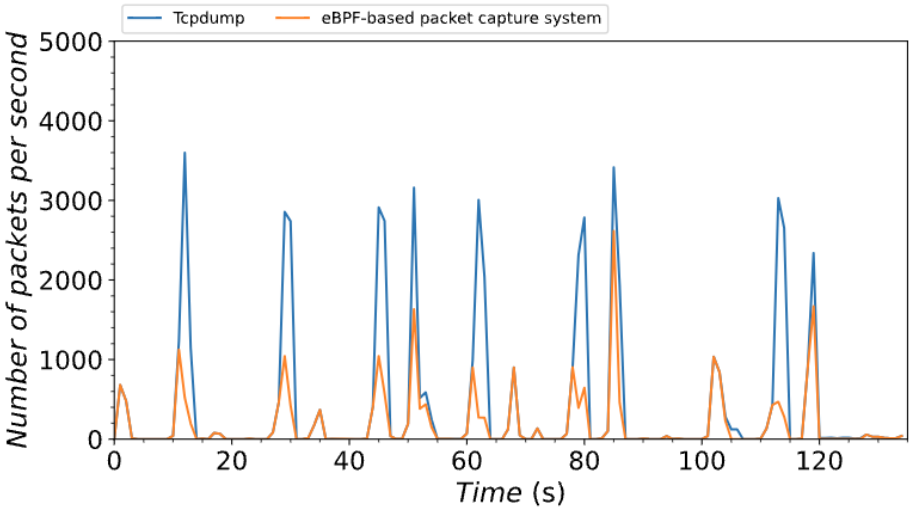

Figure 5: Packet capture performance of the Python-based prototype implementation

were dropped by the proposed system. Overall, approximately 55.61% of the total packets were dropped. This loss occurs because data written to the perf ring buffer in kernel space is overwritten before it can be read into user space. While our proposed system stores application metadata of up to 65 bytes in the perf ring buffer alongside each packet's data, this additional data has minimal impact on the 1 MB perf ring buffer capacity. Rather, our analysis suggests that the processing overhead in user space is the primary performance bottleneck. These findings indicate that the capture performance of prototype implementation requires further optimization.

We also evaluate the resource utilization of the proposed system in terms of CPU and memory overhead. During generating network traffic at a fixed packet rate using *hping3*, CPU utilization and memory usage were recorded at one-second intervals with *vmstat* command. To take into account the influence of other processes on the measurement, the proposed system starts 10 seconds after *vmstat* starts monitoring. This delay allows for a clear assessment of the overhead introduced by the proposed system and the eBPF-based packet processing. The experiment was conducted with minimum-sized UDP packets at seven distinct packet transmission rates: 1, 10, 100, 1k, 10k, 100k, and 1Mpps.

Figure 6 presents the CPU impact of the proposed system. The x-axis represents the elapsed time since the experiment started, while the y-axis shows the CPU utilization percentage. Each line corresponds to a different packet transmission rate.

As shown in the figure, all transmission rates exhibited similar growth patterns during the initial phase. CPU utilization before the proposed system started corresponds the CPU load caused by *hping3* depending on the transmission rate. After the system was launched at 10 seconds, CPU utilization increased by 11 to 14 percentage points. Packet capture began at 18 seconds after the system's bootstrapping process, and CPU utilization varied according to the transmission rate. We compare CPU utilization at 5 and 20 seconds. At 5 seconds, only *hping3* is running, while at 20 seconds, CPU utilization has stabilized after packet capture began. The maximum difference between 5 and 20 seconds was recorded at a transmission rate of 1 kpps. CPU utilization at 5 seconds was 1%, while at 20 seconds, it was 23%, resulting in difference of 22 percentage points.

The variation in CPU utilization across different transmission rates is due to the write speed of the perf ring buffer. In particular, when transmission rates exceed 100 pps, writes to the perf ring buffer increase significantly, causing CPU utilization to rise sharply. This increased write activity leads to buffer overwrite operations, which we identify as a primary cause of packet loss, as discussed in our packet capture performance analysis.

Figure 7 presents the memory impact of the proposed system. The x-axis represents the elapsed time, and the y-axis shows the memory utilization percentage. Each line corresponds to a different

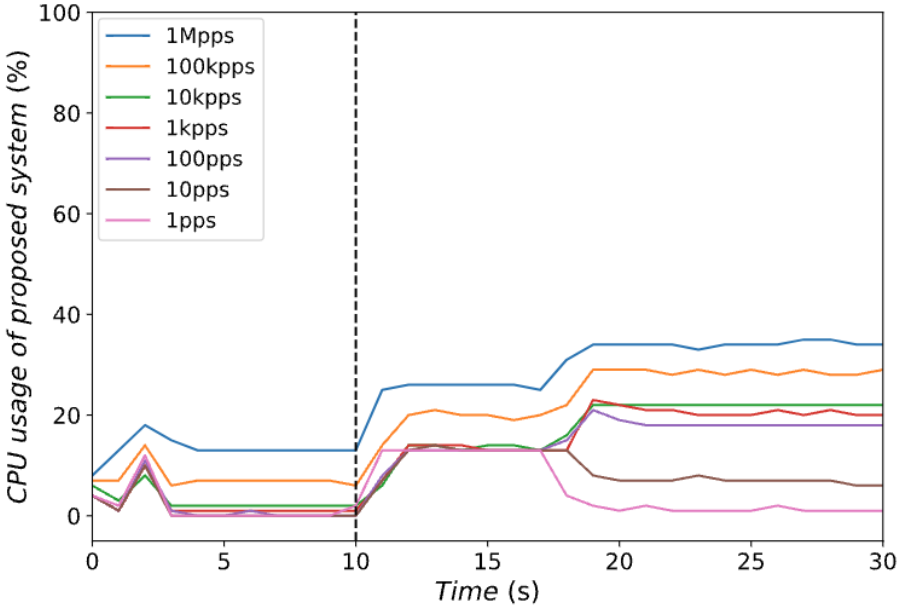

Figure 6: CPU utilization of the proposed system

packet transmission rate.

As shown in the figure, the memory usage due to system startup is observed from 10 to 18 seconds, similar to the CPU utilization results. Subsequently, memory utilization varies according to the transmission rate. Analysis of memory consumption during system initialization revealed that the proposed system consumed 2.6 to 3.1 percentage points of memory. When the packet transmission rate exceeds 1 kpps, a continuous increase in memory utilization is observed over time. The memory utilization was 21.69% at 20 seconds and 22.05% at 30 seconds, with a difference of approximately 0.36 percentage points. This is due to the front-end program's processing being unable to keep up with the write rate of the perf ring buffer, resulting in processing delays.

The evaluation results reveal that while the system maintains partial packet capture capability throughout operation, it exhibits performance limitations in handling typical Web traffic loads. To identify the source of packet loss, we investigated the packet recording and extraction processes in the perf ring buffer. Even when packet losses occur, the lost count [35] remains zero at the packet recording process within the kernel space. This result suggests that packets are lost in user space even though the kernel space successfully writes every packet to the buffer. This detailed analysis indicates that optimization efforts should focus on improving user space processing efficiency. Furthermore, the observed maximum increase of 22 percentage points in CPU utilization needs further investigation into its correlation with specific aspects of the prototype implementation.

4.3 Performance improvements with enhanced implementation

To investigate the cause of high CPU utilization and performance bottlenecks in the prototype implementation, parts of the modules were re-implemented in Go, a systems programming language. This re-implementation aimed to determine whether the high CPU utilization and low capture performance of the prototype implementation were due to characteristics of implementation language or inherent limitations in the use of eBPF technology.

The eBPF-based functionality in the re-implemented system was developed using *gobpf* [36], a Go package for eBPF. The packet handler and application resolver, which are responsible for packet capture functionality, were re-implemented and evaluated under the same experimental conditions described in the previous section. Figure 8 presents the experimental results of capturing web browsing traffic.

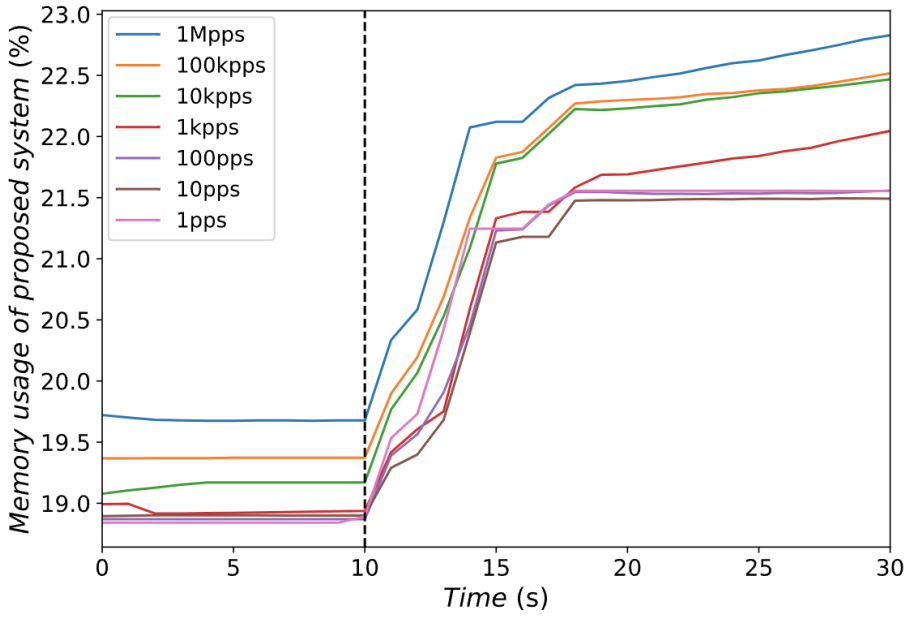

Figure 7: Memory usage of the proposed system

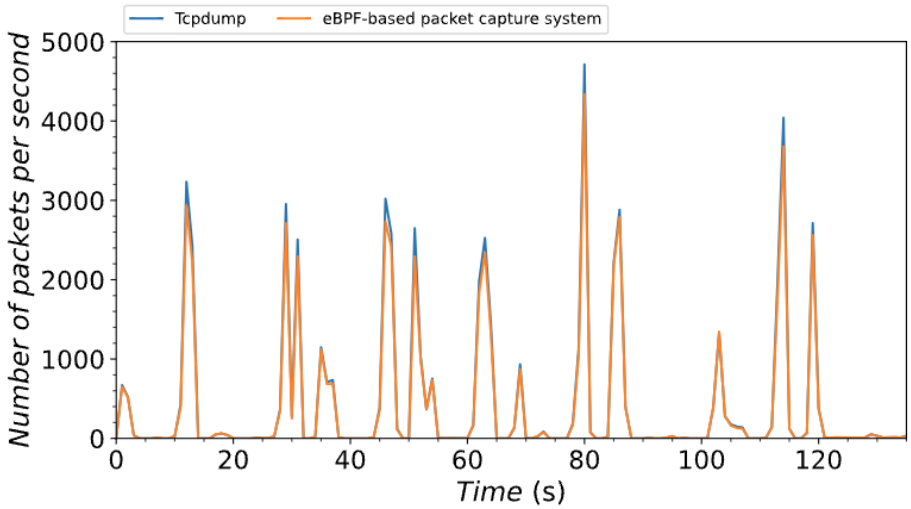

Figure 8: Packet capture performance of the Go-based prototype implementation

As shown in Figure 8, the Go-based implementation exhibited performance comparable to tcpdump when the packet rate ranged from approximately 2,500 to 4,000 packets. The packet loss rate was 7.60% of the total observed packets, which represents a significant improvement over the 55.61% loss rate observed in the Python-based implementation.

While the previous evaluations used web browsing traffic to simulate real-world conditions, packet processing capability is typically evaluated by measuring how long a system can continuously observe minimum-sized packets [8]. Therefore, to determine the capture performance limits under fixed packet transmission rates, additional evaluations were conducted for Go-based implementation. Figure 9 presents a scatter plot comparing packet capture performance between our proposed system and tcpdump. We evaluated the performance by sending UDP packets with minimum size at seven fixed rates ranging from 1 pps to 1 Mpps. The x-axis represents the number of packets captured by tcpdump, while the y-axis shows the number of packets captured by our proposed system.

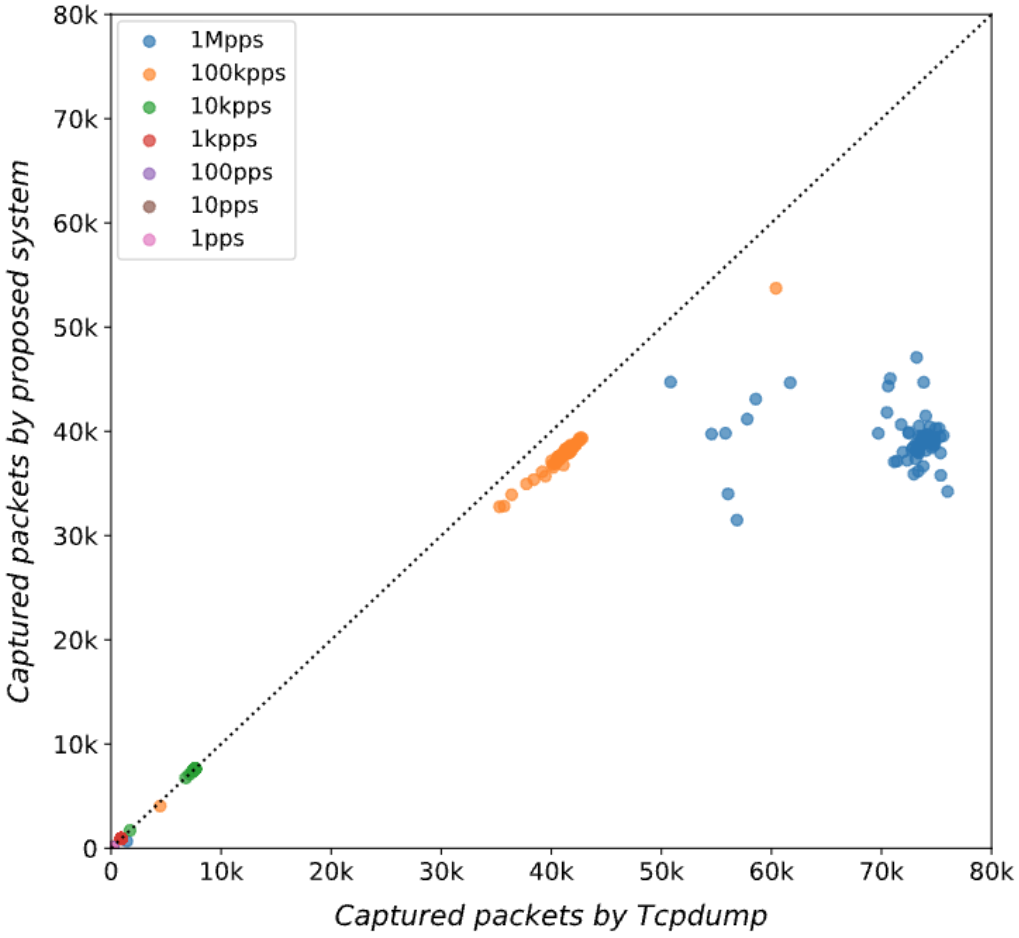

Figure 9: Fixed-rate capture performance (Go-based implementation - 1 MB perf ring buffer)

As shown in the figure, the proposed system maintains complete packet capture capability up to 10 kpps, with performance gradually degrading around 40 kpps. Notably, at a transmission rate of 100 kpps, the number of captured packets is lower than that of tcpdump. Furthermore, at a transmission rate of 1 Mpps, while tcpdump observed more than 70k packets, the proposed system only captured approximately 40k packets. These results indicate that the current Go-based implementation of the proposed system has an upper limit of approximately 40 kpps for packet capture.

To further validate that the factors contributing to the poor capture performance of the Python-

based implementation, disk I/O was evaluated by measuring the iowait value. This metric represents the time the CPU spends waiting for responses from I/O devices such as disks and networks. The iowait values were measured at one-second intervals using the *mpstat* command. Packet transmission, controlled using the *hping3* command, was initiated at around the five-second mark and maintained at a fixed rate of 10 kpps. This rate corresponds to the threshold at which significant packet loss begins to occur in the in the Python-based implementation. This experiment was conducted for three capture systems: the Python-based implementation, the Go-based implementation, and tcpdump.

Figures 10, 11, and 12 show the progression of iowait values for each CPU core. The x-axis represents the elapsed time since the experiment started and the y-axis shows the iowait values.

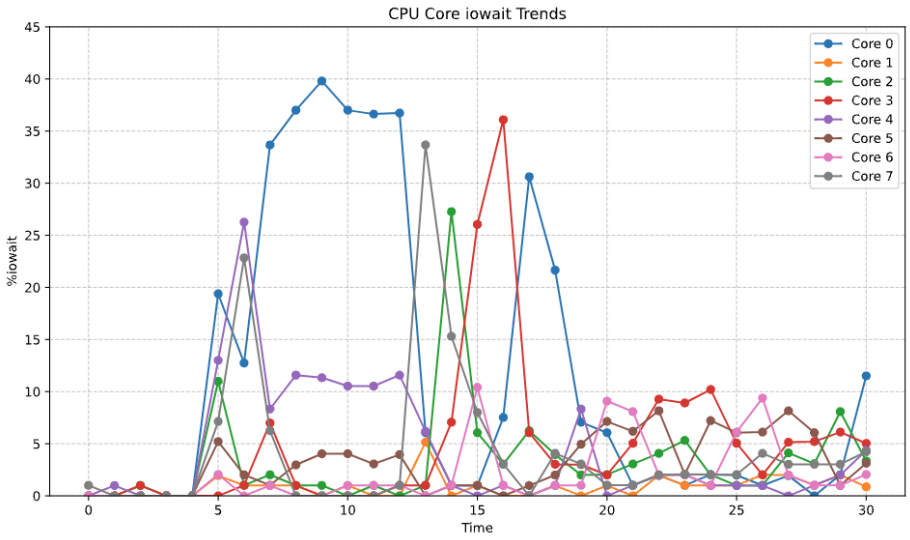

Figure 10: Per-core iowait measurements of the Python-based implementation

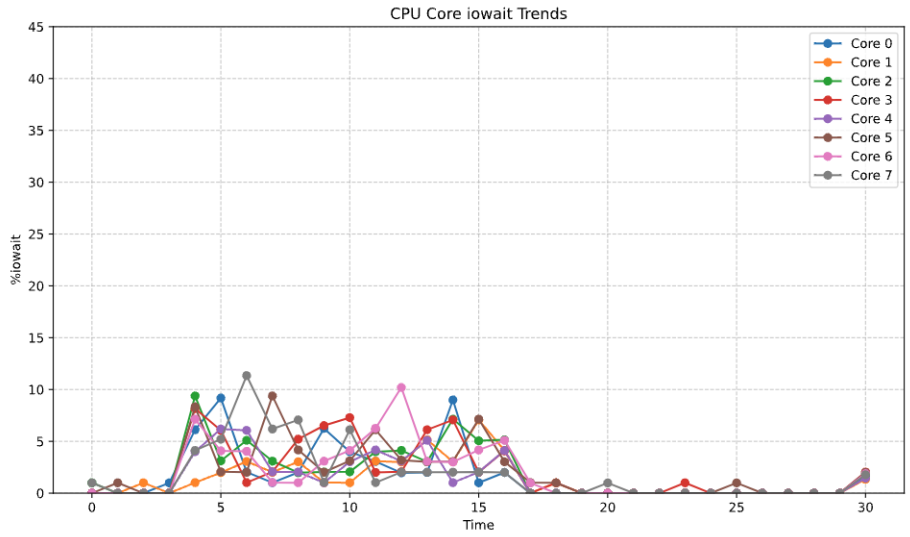

Figure 11: Per-core iowait measurements of the Go-based implementation

In the Python-based implementation, the iowait value of core 0 increases up to 40%, while other cores maintain low values. This indicates that packet processing is concentrated on a single core. In contrast, the Go-based implementation generally keeps iowait values below 12% for all cores,

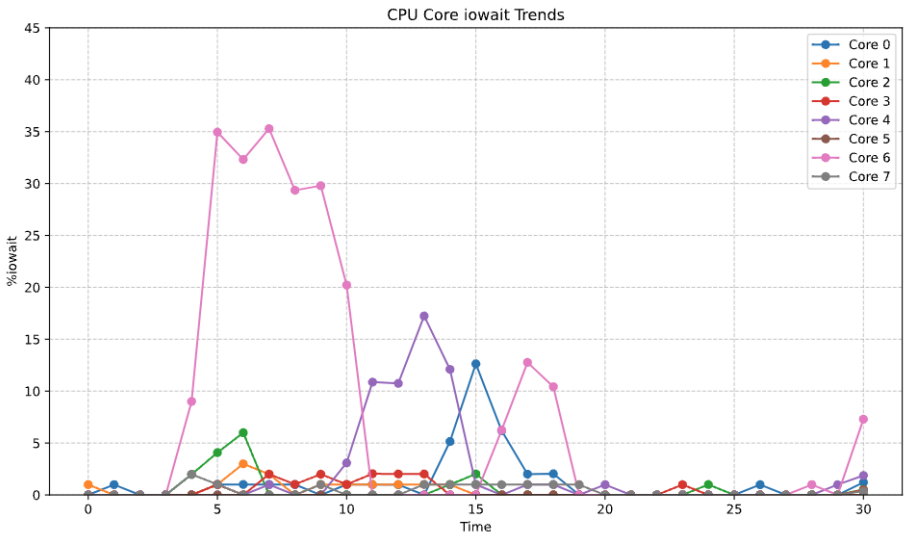

Figure 12: Per-core iowait measurements of tcpdump

suggesting that processing is distributed across multiple cores. For tcpdump, which is used as a comparison baseline, it showed stable packet capture performance while maintaining an iowait value of approximately 35% on a single CPU core.

These experimental results reveal that the low processing performance of the Python-based implementation is due to its single-threaded execution. This constraint stems from Python's Global Interpreter Lock (GIL) [37], a mechanism that restricts program execution to a single thread. While disabling the GIL or migrating from threading to multiprocessing could potentially mitigate this issue, the current Python's inherent limitations in high-speed processing make it unsuitable for packet capturing in practical scenarios requiring stable, high-speed performance.

Despite the improvements in the Go-based implementation, some packet loss was still observed, highlighting the need for further analysis.

Further performance improvements in the proposed system are possible with the additional perf ring buffer allocation. The perf ring buffer size used in the experiments is specified in 4 KB units according to the page size defined by the toolkit's functionality [38, 35]. In the next experiment, we configured the perf ring buffer to its maximum allowable size of 65,536 pages (equivalent to 262 MB) and evaluated the performance. The capture performance was evaluated while UDP traffic at the maximum rate with minimum-sized packets using *hping3*'s flood mode. Figure 13 presents a time-series graph depicting the number of packets captured by both tools. The x-axis represents the elapsed time since the capture started, while the y-axis shows the number of packets captured.

As shown in the figure, the Go-based implementation achieved a maximum packet capture rate of approximately 200 kpps, which is equivalent to tcpdump's performance. However, we observed that our system experienced performance degradation at around 20, 40, and 55 seconds. We suspect that this degradation was due to resource exhaustion on the Linux VM because tcpdump's performance also declined shortly after our system's degradation.

This result confirms that the additional perf ring buffer allocation significantly improves capture performance. However, the front-end program was overwhelmed by the high packet volume, causing the processing delays similar to those observed in the Python-based implementation. While the packet group was extracted from the perf ring buffer, the processing could not be completed before the next perf ring buffer read. Addressing these limitations and improving the front-end program's efficiency will be a focus of future work.

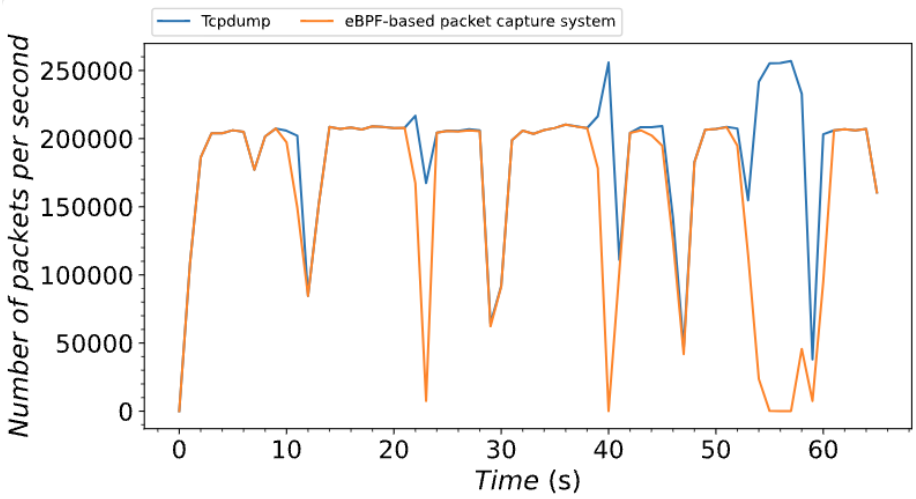

Figure 13: Capture performance of Go-based implementation with maximum ring buffer size

4.4 Discussion

As confirmed in the evaluation section, our current prototype implementation, using a 1 MB perf ring buffer, achieves a capture performance of 40 kpps. While this is theoretically sufficient for networks with bandwidths up to approximately 20 Mbps, performance optimization is still required to expand its applicability in versatile environments. The evaluation revealed that stable packet capture depends on both an appropriately sized perf ring buffer and efficient processing in the front-end program. Therefore, it is essential to determine a suitable perf ring buffer size that ensures stable packet capture without causing processing delays in the front-end program.

The additional CPU and memory overhead introduced by the proposed system raises security concerns. This performance burden can degrade the throughput of security-related processes, such as intrusion detection systems or anti-malware software, potentially reducing the accuracy of security monitoring. Such performance degradation might provide an opportunity for attackers or malware to evade detection or exploit system vulnerabilities. Moreover, because the overhead increases with the number of packets captured, the system may become more vulnerable to flooding-based denial-of-service attacks. In these situations, even a modest amount of attack traffic could significantly impact the system's functionality. Consequently, evaluating the security impact and reducing the overhead are important directions for future work.

For performance enhancement and overhead reduction, there is room for architectural improvements in the proposed system. Currently, the system captures packets and retrieves application-related information before storing them together in a sequential manner. However, this approach introduces processing delays and degrades packet capture performance. To mitigate this issue, packet capture and the collection of process-related information, including application metadata, should be decoupled and executed in parallel. The outputs from these independent processes can then be combined later for storage. This approach maintains accurate relationships between information sources while reducing the processing overhead and delay. Furthermore, it enables effective integration with existing process monitoring software, potentially enhancing forensic investigation methods.

Despite performance concerns in the current implementation, embedding application metadata in packet data offers two key advantages for forensic analysis: reliable application identification even in encrypted communications and the feasibility of cross-referencing across multiple information sources.

First, reliable application identification minimizes reliance on inference-based techniques. Conventional approaches require Deep Packet Inspection (DPI) or machine learning-based estimation [7] to determine which applications generated specific network traffic. DPI is ineffective for encrypted traffic and raises privacy concerns, while machine learning approaches require substantial

computational resources, training data, and processing time. Our proposed system directly embeds application metadata with each packet, providing efficient and reliable identification without relying on content inspection or statistical inference.

Second, our system enables cross-referencing between application logs and packet data. Reliable network forensics requires integration of multiple information sources [3, 39]. However, conventional network forensics is limited by the absence of source application within packet data itself, making cross-referencing challenging. Our system addresses this issue by directly embedding application metadata into packet data, establishing clear correspondence between these two information sources. As a result, this feature enhances the reliability of forensic analysis by facilitating a more in-depth understanding of each data source and enabling the detection of inconsistencies between them – capabilities that are difficult to achieve when relying on a single source.

5 Conclusion

In this paper, we proposed an eBPF-based packet capture system that embeds application metadata in PCAPNG file format usable by existing packet analyzers such as Wireshark. It can retain the correlation between packet data and application processes that generate the packet in a single file, thereby making it more useful for forensic analysis.

Through prototype implementation in Python and experimentation, the proof of concept for the proposed system was verified, and performance-related challenges were identified. These challenges were further analyzed using an enhanced implementation in the Go language. The Go-based implementation demonstrated better packet capture performance than Python-based implementation and showed that performance can be further improved by appropriately configuring the perf ring buffer size. Consequently, we conclude that our proposed approach will contribute to improving the efficiency of network forensics.

In future work, we aim to investigate the optimal perf ring buffer size for eBPF-based packet monitoring to enhance system practicality. Although the current implementation is limited to Linux distributions, we will explore ways to extend support to IoT devices and other operating systems to enable broader deployment across diverse computing environments. Ensuring data integrity, which is essential for maintaining evidential reliability in forensic investigations, also requires further consideration.

Moreover, correlating packets with application metadata using eBPF offers significant value for various security applications. One key area is access control within the Zero Trust Security framework. Zero Trust requires thorough verification and fine-grained control over all communications. Ideally, access decisions should be based not only on source and destination hosts or ports but also on the specific application and user involved in the interaction [40]. The correlation between packets and application metadata provided by our system serves as a reliable foundation for enforcing such context-aware, fine-grained access policies. We will explore how to leverage eBPF's capabilities to integrate this correlation process into strict access control mechanisms based on user identity, application activity, and service usage.

Acknowledgement

The authors are grateful to Dr. Keeni Glenn Mansfield for his valuable comments. This work was partially supported by the JSPS KAKENHI Grant Number JP25K15120.

References

[1] Changwei Liu, Anoop Singhal, and Duminda Wijesekera. A LOGIC-BASED NETWORK FORENSIC MODEL FOR EVIDENCE ANALYSIS. In Gilbert Peterson and Sujeet Shenoi, editors, *Advances in Digital Forensics XI*, pages 129–145, Cham, 2015. Springer International Publishing.

[2] Timothy J. Shimeall and Jonathan M. Spring. Chapter 11 - Network Analysis and Forensics. In Timothy J. Shimeall and Jonathan M. Spring, editors, *Introduction to Information Security*, pages 235–251. Syngress, Boston, 2014.

[3] Masaki Kamizono, Takashi Tomine, Yu Tsuda, Masashi Eto, Yuji Hoshizawa, and Daisuke Inoue. Proposal of Forensics Method Based on Communication Procedure of Process. *Computer Security Symposium 2014 (CSS 2014)*, 2:167–174, 2014. (in Japanese).

[4] eBPF.io. ebpf documentation. `https://ebpf.io/what-is-ebpf/`. [Accessed 10-02-2025].

[5] Michael Tüxen, Fulvio Risso, Jasper Bongertz, Gerald Combs, Guy Harris, Eelco Chaudron, and Michael Richardson. PCAP Next Generation (pcapng) Capture File Format. Internet-Draft draft-ietf-opsawg-pcapng-02, Internet Engineering Task Force, August 2024. Work in Progress.

[6] Ray Hunt and Sherali Zeadally. Network Forensics: An Analysis of Techniques, Tools, and Trends. *Computer*, 45(12):36–43, Dec 2012.

[7] Leslie F. Sikos. Packet analysis for network forensics: A comprehensive survey. *Forensic Science International: Digital Investigation*, 32:200892, 2020.

[8] Shintaro Ishihara and Toyokazu Akiyama. A Tuning Method of a Monitoring System for Network Forensics in Cloud Environment. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 01, pages 951–954, July 2018.

[9] Abount Wireshark. `https://www.wireshark.org/about.html`. [Accessed 10-02-2025].

[10] Suleman Khan, Abdullah Gani, Ainuddin Wahid Abdul Wahab, Muhammad Shiraz, and Iftikhar Ahmad. Network forensics: Review, taxonomy, and open challenges. *Journal of Network and Computer Applications*, 66:214–235, 2016.

[11] Satoshi Mimura and Ryouichi Sasaki. Proposal and Evaluation of the Preservation Method of the Network Packets Associated with Process Information. *IPSJ Journal*, 57(9):1944–1953, 2016. (in Japanese).

[12] Hirochika Asai, Kensuke Fukuda, and Hiroshi Esaki. Traffic causality graphs: Profiling network applications through temporal and spatial causality of flows. In *2011 23rd International Teletraffic Congress (ITC)*, pages 95–102, 2011.

[13] FPGA Introduction. `https://indico.ictp.it/event/a11204/session/7/contribution/4/material/0/0.pdf`. [Accessed 29-04-2025].

[14] Data Plane Development Kit (DPDK) — intel.com. `https://www.intel.com/content/www/us/en/developer/topic-technology/networking/dpdk.html?wapkw=dpdk`. [Accessed 28-04-2025].

[15] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. hXDP: Efficient software packet processing on FPGA NICs. *Commun. ACM*, 65(8):92–100, July 2022.

[16] IO Visor. XDP - eXpress Data Path. `https://www.iovisor.org/technology/xdp`. [Accessed 10-02-2025].

[17] Pedro Salva-Garcia, Ricardo Ricart-Sanchez, Emilio Chirivella-Perez, Antonio Garrido, Antonio J. Jara, and Miguel Malumbres. XDP-Based SmartNIC Hardware Performance Acceleration for Next-Generation Networks. *Journal of Network and Systems Management*, 30(4):75, 2022.

[18] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The eXpress data path: fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '18, page 54–66, New York, NY, USA, 2018. Association for Computing Machinery.

[19] TCPDUMP & LIBPCAP. `https://www.tcpdump.org/`. [Accessed 29-04-2025].

[20] Steven McCanne and Van Jacobson. The BSD packet filter: a new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, USENIX'93, page 2, USA, 1993. USENIX Association.

[21] LWN.net. A thorough introduction to eBPF. `https://lwn.net/Articles/740157/`. [Accessed 04-05-2025].

[22] The Cilium. Cilium: Networking and security for containers with BPF and XDP. `https://cilium.io/blog/2017/3/16/cilium-networking-and-security-for-containers-with-bpf-and-xdp/`. [Accessed 10-02-2025].

[23] Liz Rice. *Learning eBPF*. O'Reilly Media, Sebastopol, CA, March 2023.

[24] David Soldani, Petrit Nahi, Hami Bour, Saber Jafarizadeh, Mohammed F. Soliman, Leonardo Di Giovanna, Francesco Monaco, Giuseppe Ognibene, and Fulvio Risso. eBPF: A New Approach to Cloud-Native Observability, Networking and Security for Current (5G) and Future Mobile Networks (6G and Beyond). *IEEE Access*, 11:57174–57202, 2023.

[25] Lei Zhang, Hui Li, Jingguo Ge, Yulei Wu, Liangxiong Li, Bingzhen Wu, and Haojiang Deng. EDP: An eBPF-based Dynamic Perimeter for SDP in Data Center. In *2022 23rd Asia-Pacific Network Operations and Management Symposium (APNOMS)*, pages 01–06, 2022.

[26] Lars Wüstrich, Markus Schacherbauer, Markus Budeus, Dominik Freiherr von Künßberg, Sebastian Gallenmüller, Marc-Oliver Pahl, and Georg Carle. Network Profiles for Detecting Application-Characteristic Behavior Using Linux eBPF. In *Proceedings of the 1st Workshop on EBPF and Kernel Extensions*, eBPF '23, page 8–14, New York, NY, USA, 2023. Association for Computing Machinery.

[27] Chang Liu, Byungchul Tak, and Long Wang. Understanding Performance of eBPF Maps. In *Proceedings of the ACM SIGCOMM 2024 Workshop on EBPF and Kernel Extensions*, eBPF '24, page 9–15, New York, NY, USA, 2024. Association for Computing Machinery.

[28] Milo Craun, Khizar Hussain, Uddhav Gautam, Zhengjie Ji, Tanuj Rao, and Dan Williams. Eliminating eBPF Tracing Overhead on Untraced Processes. In *Proceedings of the ACM SIGCOMM 2024 Workshop on EBPF and Kernel Extensions*, eBPF '24, page 16–22, New York, NY, USA, 2024. Association for Computing Machinery.

[29] The Linux Kernel documentation. Perf ring buffer. `https://docs.kernel.org/userspace-api/perf_ring_buffer.html`. [Accessed 15-02-2025].

[30] GitHub - iovisor/bcc: BCC - Tools for BPF-based Linux IO analysis, networking, monitoring, and more — github.com. `https://github.com/iovisor/bcc`. [Accessed 10-02-2025].

[31] Dylan Reimerink. Maps. `https://docs.ebpf.io/linux/concepts/maps/`. [Accessed 10-02-2025].

[32] GitHub - giampaolo/psutil: Cross-platform lib for process and system monitoring in Python — github.com. `https://github.com/giampaolo/psutil`. [Accessed 10-02-2025].

[33] GitHub - rshk/python-pcapng: Pure-Python library to parse the pcap-ng format used by newer versions of dumpcap & similar tools. — github.com. https://github.com/rshk/python-pcapng. [Accessed 10-02-2025].

[34] Selenium. The Selenium Browser Automation Project. https://www.selenium.dev/documentation/. [Accessed 12-02-2025].

[35] bcc Reference Guide. https://github.com/iovisor/bcc/blob/master/docs/reference_guide.md. [Accessed 10-02-2025].

[36] GitHub - iovisor/gobpf: Go bindings for creating BPF programs. — github.com. [Accessed 10-02-2025].

[37] python. GlobalInterpreterLock. https://wiki.python.org/moin/GlobalInterpreterLock. [Accessed 15-02-2025].

[38] Red Hat Documentation. Chapter 14. Large Memory Optimization, Big Pages, and Huge Pages. https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/5/html/tuning_and_optimizing_red_hat_enterprise_linux_for_oracle_9i_and_10g_databases/chap-oracle_9i_and_10g_tuning_guide-large_memory_optimization_big_pages_and_huge_pages. [Accessed 15-02-2025].

[39] Chen Lin, Li Zhitang, and Gao Cuixia. Automated Analysis of Multi-Source Logs for Network Forensics. In *2009 First International Workshop on Education Technology and Computer Science*, volume 1, pages 660–664, 2009.

[40] Scott Rose, Oliver Borchert, Stuart Mitchell, and Sean Connelly. Zero Trust Architecture. Special Publication (NIST SP), National Institute of Standards and Technology, Gaithersburg, MD, https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=930420, 2020-08-10 04:08:00 2020. [Accessed 02-05-2025].