XPLOG: A Dynamic Observability Framework for Distributed Sandboxed Microservices

Utkalika Satapathy, *Member, IEEE*, Harsh Borse, Rajat Bachhawat, Neha Dalmia, Subhrendu Chattopadhyay, and Sandip Chakraborty, *Senior Member, IEEE*

***Abstract*—Runtime application observability is crucial not only for system provenance but also for the orchestration of deployed microservices in dynamic sandboxed distributed computing environments. Also, log extraction and aggregation in highly distributed and sandboxed environments pose significant challenges, especially when preserving the causal order of the events triggered by different asynchronous microservices running over multiple hosts. However, ensuring causally consistent logging of application events is challenging, although it is vital for continuously tracing and profiling the underlying platform. This paper proposes XPLOG, a scalable, pluggable, easily deployable, and dynamic runtime observability framework for distributed sandboxed computing platforms that leverages the capability of extended Berkeley Packet Filters (eBPF) to intercept system-level events within the host while capturing and amalgamating relevant application and system logs to produce globally causally-consistent log streams. Through qualitative and quantitative analysis, we observe that XPLOG significantly improves log richness with minimum system overhead while preserving the causality of log-generating events across multiple microservices.**

***Index Terms*—Runtime observability, causally-consistent logs, distributed provenance tracking.**

I. Introduction

CLOUD-NATIVE technologies have transformed application development by enabling scalable, dynamic, and distributed IT infrastructure. Built on core components such as containers, microservices, and immutable infrastructure, these architectures create a robust, maintainable, and loosely coupled ecosystem that, when combined with automation, allows frequent and predictable changes with minimal effort. The *Cloud Native Computing Foundation*[1] (CNCF) aims to make cloud-native computing ubiquitous and accessible across industries. In particular, integrating microservice architecture with the DevOps philosophy has significantly improved application development, deployment, and management. Unlike monolithic applications, microservices divide functionality into independent, manageable, and easily deployable components, driving a widespread shift from monolithic to microservice-based designs. However, this distributed paradigm also introduces new operational challenges. As IT infrastructure becomes increasingly heterogeneous, spanning edge computing, cloud-native deployments, and Industrial Internet of Things (IIoT) configurations, comprehensive monitoring and observability are essential. While monitoring tracks predefined metrics, logs, and symptoms, observability [1], [2] provides the contextual understanding required to investigate and identify root causes by analyzing system behavior and interactions. Traditional monitoring and observability tools are often inadequate for the dynamic nature of microservices and rapid deployment cycles, underscoring the need for advanced observability frameworks in cloud-native distributed environments to improve root cause analysis and reduce metrics such as *Mean Time To Repair* or *Mean Time To Recovery* (MTTR).

Building on the challenges of monitoring and observability in distributed cloud-native systems, general-purpose syscall telemetry frameworks such as Sysdig [3], SysFlow [4], and eAudit [5] provide valuable system-level visibility but fall short in capturing distributed causality and bridging application- and OS-layer contexts. For instance, Sysdig [3] enables rich per-host syscall tracing with container awareness, yet it lacks inter-host ordering guarantees and relies only on timestamps. SysFlow [4] addresses log volume by summarizing syscall data into flow records, but this aggregation discards fine-grained causal relationships, limiting its effectiveness for deep debugging and root cause analysis in multi-host deployments. Similarly, eAudit [5] supports comprehensive syscall monitoring, but its scope remains confined to single-host OS-level auditing without container or cloud-native awareness. These limitations highlight the need for an observability framework that preserves both intra- and inter-host causality while enabling end-to-end tracing of event chains across microservices.

One critical requirement for developing an observability framework is to collate the log messages from running microservices and the underlying host OS in a meaningful manner, which not only captures the events executed by various microservices but also should preserve their causal or execution order [6], [7]. These collated order-preserving logs, called *causally-consistent log stream*, help better analyze the root cause. For runtime observability, the platform can apply an event-sequence-based

Received 20 February 2025; revised 15 August 2025; accepted 11 September 2025. Date of publication 13 October 2025; date of current version 5 February 2026. This work was supported by the Indira Gandhi Center for Atomic Research (IGCAR). *(Corresponding author: Sandip Chakraborty.)*

Utkalika Satapathy, Harsh Borse, and Sandip Chakraborty are with the Department of Computer Science and Engineering, Indian Institute of Technology, Kharagpur 721302, India (e-mail: utkalika.satapathy01@gmail.com; harshzf2@gmail.com; sandipc@cse.iitkgp.ac.in).

Rajat Bachhawat and Neha Dalmia are with Quadeye, Gurugram 122002, India (e-mail: rbachhawat.96@gmail.com; nehadalmia002@gmail.com).

Subhrendu Chattopadhyay is with the Thapar Institute of Engineering and Technology (TIET), Patiala 147004, India (e-mail: subhrendu.subho@gmail.com).

Digital Object Identifier 10.1109/TSC.2025.3612900

[1]https://www.cncf.io/ (Accessed: 2025/10/03 10:50:39)

filter to the generated log streams, which can extract the necessary information more quickly without needing to parse the whole log. Also, the causally consistent log messages help to create the provenance graph [8], [9] dynamically, which can further be used for system vulnerability detection [10], [11], root-cause analysis [12], [13], [14], etc. Our paper aims to address this gap by exploring the open research challenges related to observability in distributed microservice architecture.

Research Challenges: Designing an observability framework over a distributed sandboxing environment (i.e., mostly lightweight containers) has the following challenges.

• Challenge ①. *Relevant Log Extraction for Multi-Host Sandboxed Applications:* Sandboxing platforms (e.g., Docker, Kubernetes) abstract application execution, making it difficult to access fine-grained application behavior. Existing solutions (e.g., `Dyninst` [15], `Angr` [16], [17]) rely on static or dynamic binary analysis to extract relevant logging information. However, such methods are impractical in distributed environments because (i) application binaries are often proprietary or inaccessible in third-party or multi-tenant deployments, and (ii) binary analysis is architecture-dependent, requiring knowledge of instruction sets or program formats. Although intermediate representations (e.g., LLVM IR, VEX) can aid cross-platform analysis, deploying and maintaining these at scale across heterogeneous hosts is infeasible. Thus, rather than inspecting application internals, we need a grey-box approach that extracts meaningful logs directly from running microservices, relying solely on application-generated logs without visibility into source code.

• Challenge ②. *Partial Visibility of Application and System Logs:* The sandboxing environment uses separate namespaces from the base kernel; The use of separate namespaces in sandboxing environments isolates process hierarchies from the base kernel, leading to fragmented log visibility. Existing tools such as `OmegaLog`, `Logstash`, and `Fluentd` [1], [18], [19] can either access base kernel system logs (when executed on the host) or container-specific application logs (when executed within the container), but not both. Consequently, system logs often lack application-specific context, such as user interactions or thread behaviors, and application logs fail to reflect underlying resource usage or kernel-level interactions. This separation necessitates independent configuration of logging tools on both the host and containers, resulting in limited visibility. This separation creates a partial view of the system, making it challenging to correlate logs across namespaces.

• Challenge ③. *Preserving Causal Consistency Across Event Logs:* One crucial aspect of system observability is to meaningfully combine the application logs with the system-generated logs to preserve each application thread's context and resource access patterns over the host OS kernel. However, combining application and system logs in a distributed microservice environment requires maintaining causal consistency, which is challenging due to parallel execution and heterogeneity across hosts. Therefore, we must combine the logs generated from the running microservices with the logs from the host OS kernel. Traditional ways of aggregating logs based on timestamps do not work for the following reasons. (i) The system timestamp is typically taken from `CLOCK_MONOTONIC` (the absolute elapsed wall-clock time since the system boot), which is CPU core-specific; therefore, containers pinned in different cores may provide different timestamps [20], [21]. (ii) The synchronization primitives may fail to preserve the ordering of the events when collating logs generated from different user and kernel threads. For example, the threads that collect logs from individual microservices may need to use a lock while collating the log messages to avoid write-write synchronization conflicts. However, the locking order may not align with the log generation order, potentially resulting in out-of-sequence logs. (iii) Modern systems are capable of ensuring intra-host log consistency to some extent [22], [23], but distributed setups introduce challenges such as inter-host synchronization and event ordering.

Therefore, these challenges highlight the need to develop advanced techniques that preserve causal consistency in distributed environments, enabling meaningful observability and effective provenance analysis.

Our Contributions: This paper presents `XPLOG`, a platform-agnostic dynamic observability framework that does not require application or platform instrumentation and can be used for both runtime analysis of the system behavior as well as periodic or offline system provenance analysis. Our key contributions to this paper are as follows.

• Contribution ①. *Development of a grey-box observability framework for multi-host sandboxed applications:* We develop the first-of-its-kind observability framework for sandboxed distributed platforms. It monitors containerized microservices running across multiple physical or virtual hosts, generating causally-consistent, context-aware logs without requiring application or platform instrumentation. At its core, `XPLOG` uses eBPF [24], [25], [26], [27] to inject customized byte-codes to specific kernel hook points for monitoring OS-level events with zero modification in the OS kernel.

• Contribution ②. *Causality-Aware Collation of System and Application Logs:* We develop novel methods for preserving causality during the log collection from multiple sandboxed microservices running over multiple hosts while handling the abovementioned challenges. The core of our approach utilizes eBPFring buffers to develop a method for making the syscalls atomic. It thus ensures that all the log messages relevant to a syscall are observed together at the generated log streams. At runtime, it dynamically maps application logs to syscall logs in an application-agnostic manner, ensuring consistent causally-ordered log generation.

• Contribution ③. *Implementation and thorough evaluation:* We open-source the implementation of `XPLOG`, evaluate it with a benchmark `DeathStarBench` [32] social media application with 30 microservices, each with 3–5 replicas, spawning over 10-30 different hosts, and compare its performance with `Tracee` [33], an eBPF-based widely-used enterprise application for platform log management. We observe that `XPLOG`-generated log reduces the disorders among the log entries and generates a causally-ordered log for parallelly running microservices with varying numbers (10,000 to 30,000) of concurrent requests from different service endpoints the benchmark provides. The logs (ordered logs) generated are comparable to those obtained from running the microservices sequentially on a single host. Further, we observe that `XPLOG` consumes minimal resources when running as a background logging service while

providing significantly more contextual information, which can help in efficient system provenance.

II. Related Work

Several research efforts have explored various aspects of provenance data extraction, provenance tracking, audit logging, log analysis, and causality analysis.

A. Logging in Distributed Cloud Infrastructure

Existing system logging approaches primarily focus on collecting effective logs for system provenance [34], [35], [36], [37]. Many of these rely on custom kernel modules—either static [1], [38], [39], proxy-based [2], [40], or Loadable Kernel Module (LKM)-based [41]—to merge application and kernel logs into provenance graphs. Static hooking requires recompiling the entire kernel, making it fragile and hard to maintain, while proxy and LKM-based methods attach only at specific hook points, limiting their scalability in large microservice deployments. LKM-based solutions are also brittle against kernel upgrades [42]. Beyond kernel hooks, several works attempt to enhance system observability through log processing and anomaly detection. For example, *SLEUTH* [43] reconstructs attack scenarios in real time using memory-based dependency graphs of audit logs, [44] combines log parsing and machine learning to detect runtime problems in console logs, and *ELT* [45] leverages fine- and coarse-grained log features for anomaly detection.

Provenance tracking has also been widely studied for both monolithic [1], [38], [40], [46] and microservice-based applications [2], [39], [41]. Approaches such as $PROV_{2R}$ [47] capture provenance from unstructured processes via record/replay and taint tracking, though they require instrumentation; *Scippa* [48] extends Android IPC to detect attacks; *Hi-Fi* [49] leverages the Linux Security Modules (LSM) framework for whole-system provenance; [50] introduces document provenance in cloud environments; and *SmartProvenance* [51] employs blockchain and smart contracts for privacy-preserving, access-controlled provenance with trust enforcement. Despite these advances, challenges remain, including dependency explosion—where excessive dependencies complicate analysis and increase overhead—and instrumentation requirements, which are resource-intensive and impractical for enterprise deployment. Recent works on causality [28], [31], [52], [53], [54] attempt to go beyond provenance by linking events, but most rely only on temporal relationships rather than true causal inference. For instance, Falcon [31] analyzes the context of network requests but lacks content-aware analysis of storage I/O, while *CauseInfer* [55] focuses mainly on numerical data and cannot handle non-numerical information such as log events or configuration data.

B. Observability in Distributed Systems

The lack of visibility into microservice applications running as containers in distributed cloud platforms is a persistent challenge. This has led to the development of various tools, from observability framework to log management and analysis solutions for cloud infrastructure and containers *Datadog* [56], *Dynatrace* [57], *New Relics* [58], *Aqua* [59] to enhance observability and debugging. However, these solutions majorly collect the logs at a high level and lack the granularity needed to capture the lower-level system events within the distributed environment. Moreover, these tools are licensed and are neither open-source nor cost-effective. Also, research work like *lprof* [60] and *Stitch* [61] uses runtime application logs to reconstruct the execution flow of requests in distributed applications. However, lprof performs static binary analysis, and Stitch relies on structured application logs with instrumentation in the code for analysis. They are more suitable choice for log management rather than log analysis because the collected logs lack context and causality.

C. Use of eBPFin Logging

On the other hand, eBPFhas been widely studied to extend the kernel functionality from several perspectives, like for fast packet processing [62], network function management [63], accelerating distributed protocols [64], in-kernel storage management [65], path-aware TCP [66], container network observability [29], [67], etc. Given its wide-scale adaptability for various applications on system observability, eBPFcan meet the requirements for the development of a fast logging mechanism for system provenance by dynamically deploying an observability module to combine the application logs from the microservices and the kernel system logs to generate the provenance graph on the fly. As discussed earlier, widely adopted state-of-the-art logging mechanisms like fluentd [19], Logstash [18], `Tracee` [33], *Tetragon* [68] etc., are not suitable for multi-layer logging. Whereas several works in the literature have focused on efficient system auditing [5], [40], [69], they have primarily focused on developing a logging mechanism for a single-host monolithic application. There have been several works in the literature [70], [71], [72], which have focused on distributed logging of events; however, they use specific techniques and resources like in-memory databases, far memory and remote memory architecture, etc., and are not suitable for sandboxed applications. Further, these tools fail to generate causally-consistent logs from multiple microservices. Notably, this is one of the vital requirements for runtime provenance over a highly distributed microservice architecture, which our proposed observability framework handles.

III. Problem Statement and System Overview

Our proposed observability framework `XPLOG` uses a distributed log collection architecture to generate causally-consistent log message streams from running microservices over sandboxed distributed platforms. In this section, we first formally define the concept of *Causally-consistent log message stream* and then discuss the broad overview of `XPLOG`.

A. Problem Statement

Let e_i and e_j be two log-generating application events (events that generate log messages, like a function/procedure call, computing loops, branching, etc.). We say e_i *happens-before* e_j, denoted as $e_i \prec e_j$, if one of the following conditions holds.

1) e_i and e_j execute on the same microservice and $T(e_i) < T(e_j)$, where $T(e)$ indicates the local clock (i.e.,`CLOCK_MONOTONIC`) reading for an event e.

TABLE I
RECENT RESEARCH WORK FOR LOGGING FRAMEWORKS COMPARED TO XPLOG

Sl No	Reference	Year	Non-invasive	Modular & Pluggable	Multi-Host Support	System-log Collection	Application-Log Collection	Real-time Log Processing	Causal Ordering	Provenance Tracking	eBPF based
1	Horus [28]	2021	✗	✗	✓	✗	✓	✗	✓	✗	✓
2	Liu et al. [29]	2020	✓	✓	✓	✓	✗	✗	✗	✗	✓
3	Viperprobe [30]	2020	✓	✗	✓	✓	✗	✗	✗	✗	✓
4	Falcon [31]	2018	✗	✗	✓	✗	✓	✗	✓	✗	✗
5	SysFlow [4]	2020	✓	✗	✗	✓	✗	✗	✗	✗	✓
6	eAudit [5]	2023	✓	✗	✗	✓	✗	✗	✗	✗	✗
7	Sysdig [3]	2016	✓	✗	✗	✓	✗	✗	✗	✗	✓
8	Our Approach (*XPLOG*)	2025	✓	✓	✓	✓	✓	✓	✓	✓	✓

TABLE II
LOG INFO CATEGORIES AND FIELDS

Category	Description	Example Fields
Event Context	Syscall-related	timestamp, datetime, syscall id, syscall name, return value
Task Context	Task which generates the syscall	host pid, host tid, host ppid, pid, tid, ppid, cgroup id, mount namespace id, pid namespace id, task command
Arguments	Syscall arguments	syscall dependent; can take upto 6 fields
Artifacts	Additional information	Executable file path, Write/Read file path
Message	App log	Log message string

2) e_i and e_j execute on different microservices and e_i triggers (*causes*) e_j (for example, e_i is a `send` call on a network socket, and e_j is the corresponding `receive` call at the other end of the socket).
3) if $e_i \prec e_j$ and $e_j \prec e_k$, then $e_i \prec e_k$.

Let $L(e)$ be the log messages corresponding to the application event e. Let $\mathcal{E}$ be the set of events generated over all the microservices for an application, and $\mathcal{L}$ be the collated log message stream from all the running microservices. We call $\mathcal{L}$ *causally-consistent* if and only if for any two application events $e_i, e_j \in \mathcal{E}$ and the corresponding log messages $L(e_i), L(e_j) \in \mathcal{L}$,

$$e_i \prec e_j \iff L(e_i) \mapsto L(e_j) \tag{1}$$

Notably, an application event e_i can generate multiple log messages (e.g., for the same event, both application logs and system logs). In this context, $L(e_k) \mapsto L(e_l)$ indicates that the entries of $L(e_k)$ come together and sequentially precede the entries of $L(e_l)$ in the generated log message stream $\mathcal{L}$. The causal-consistency also ensures that the log messages generated from multiple events do not get mixed up with each other. We next discuss the broad overview of our proposed observability framework `XPLOG` to extract the causally-consistent logs from the running microservices.

B. Basic Working Principle and System Overview

Following the general principle of microservice architecture [73], [74], we assume the application microservices deployed on each host run inside individual containers. `XPLOG` captures the system logs from individual hosts, orders them based on the causal order of application events executed over different microservices, and streams them to the downstream services in the causally-consistent order. To enrich the individual log entries, `XPLOG` adds four categories (see Table II) of information along with the log messages. Among these categories, `Event context` and `Task context` are helpful to disambiguate log entries from multiple PID namespaces. `Arguments` precisely capture the syscall parameters relevant for debugging syscalls, forensics, etc. On the other hand, `Artifacts` fields provide the executable file location and the files accessed by the caller program.

`XPLOG` utilizes eBPFto deploy the logging module within the host OS kernel in an application-agnostic manner and generates the causally-consistent log message stream $\mathcal{L}$. Notably, eBPFenables injection of customized codes called *probes* to specific kernel hook points, called the *tracepoints*, and thus can extend the capabilities of the kernel safely and efficiently at runtime without requiring any changes to the kernel source code or loading the kernel modules. To generate *causally-consistent logs* across microservices running over multiple host machines, `XPLOG` employs a two-step approach: (1) first, it ensures causal consistency across the log messages generated from microservices running over a single host, and then, (2) it preserves the causal-consistency during log collation from multiple hosts. Notably, while we can use the concept of logical clocks from the standard distributed systems literature (like a per-host *vector clock* [75]) to solve (2), the first one is much more challenging as the containers use a different namespace than the host OS kernel. Further, the application logs generated from the containerized microservices use separate thread execution contexts at various namespaces. Consequently, we use a **novel design approach** following the idea that containerized applications use syscalls to access resources over the host OS, and syscall executions are *atomic operations* inside a thread execution context. Therefore, `XPLOG` monitors syscall executions across the running threads over the host OS kernel and collates the generated log messages based on the syscall execution order. However, as the containerized applications use a different PID namespace than the host OS, there is a need to map the application processes with the corresponding host OS processes. `XPLOG` uses various eBPFdata structures to address this.

As shown in Fig. 1, `XPLOG` comprises (1) the `XPLOG` *Agent*, residing within each host, responsible for ensuring causal consistency of the log messages generated from microservices running over a single host, and (2) the `XPLOG` *Collector* that collates the log messages across the hosts while maintaining causal consistency and streams the log messages to the downstream services depending on their needs. Consequently, various platforms and application components can be attached with the `XPLOG` *collector* for extracting the relevant log messages and using them for runtime analysis. Next, we discuss the functionalities of these two components in detail.

IV. COMPONENTS OF XPLOG

The `XPLOG` Agents are deployed as privileged containers over the host machines, which allows them to monitor the microservice logs (application-level logs) from other

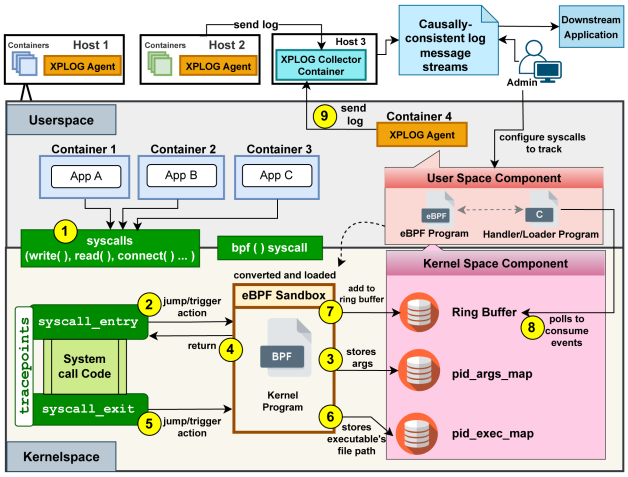


Fig. 1. XPLOG: User and Kernel-space Components.

containers. In addition, they require capabilities such as CAP_BPF, CAP_SYS_ADMIN, and CAP_PERFMON to attach eBPF probes. However, the monitored microservice containers do not need any additional privilege. The observability layer is decoupled from application logic, and it supports better operational simplicity and broader compatibility. The user-space and the kernel-space components of XPLOG Agents work as follows.

A. XPLOG Agent: User-Space Component

The user-space component of XPLOG Agent is responsible for interfacing between the Log collector and the host OS kernel. Administrators can configure the system through the user-space component to monitor specific syscalls based on their preferences. The package utilizes kernel tracepoints with eBPFprobes to monitor the host kernel-level events. To enrich individual log entries, XPLOG adds four categories of log information, as shown in Table II. Although the probe creation and other auxiliary activities are explained in Section IV-B in more detail, the user-space component is responsible for the invocation and management of these tracepoints. The generated event logs from the tracepoints are stored in an eBPFring buffer, which is shared with the user-space and minimizes the synchronization overhead. Periodically, the user-space component polls, parses, interprets, and sends the event logs to the XPLOG Collector as aggregated host-level log over TCP. Utilizing kernel tracepoints with the help of the kernel-space component as discussed next, especially syscall entry and exit events, an eBPFprobe ensures the causal order of host-level logs within the host's scope, consequently extending this causality order to the collated log message stream.

B. XPLOG Agent: Kernel-Space Component

Once the user-space component is configured correctly, it invokes the kernel-space component. The kernel-space component has several modules that work together to extract and order the log messages from both applications and the host OS while preserving the event execution order.

1) Tracepoints & eBPFProbes: We configure Kernel-space tracepoints to hook syscall entry and exit events by executing eBPFprobes. The entry probe extracts syscall arguments, while the exit probe monitors contextual information (e.g., task, container, event context) and artifacts like executable path and syscall return value. Separate probe programs are developed for each monitored syscall due to variations in the extracted information.

2) eBPFMaps: The maps are used to share data between the user and the kernel-space. We use following eBPFmaps:

i) *pid_args_map:* An eBPFmap data structure named `pid_args_map` is used to collate the extracted information from the tracepoints. Without this map, logging of entry and exit tracepoints separately increases the output log size and may impact the performance. The `pid_args_map` is indexed by thread IDs, as it is unique in the host. We designate the combination of thread ID and host PID as the XPLOG ID aiding in distinguishing host-level logging within the XPLOG log message stream. Despite the causal ordering of the XPLOG generated log, logs within the collated log message stream become interleaved across different hosts. Therefore, the absence of the XPLOG ID makes it challenging to discern logs from various hosts or containers associated with a request. Each syscall entry probe places the extracted syscall arguments as the value in the map. The syscall exit probe pops the argument from the map and generates the host-level collated syscall log. Once the host-level collated syscall log is generated, it is put into the eBPFBuffer to be polled from the user-space component. Consequently, only one log will be pushed to the eBPFbuffer at a time, ensuring sequential processing. This ensures the *causal ordering* as syscall executions are *atomic operations* inside a thread execution context. Moreover, the process ensures reutilization of map entries for different syscalls called from a single thread, which reduces the memory footprint of XPLOG.

(ii) *pid_exec_map:* Notably, extracting the absolute path of the program executable that triggered the application or system call log is necessary to ensure the causal ordering. In a complex distributed application, having the absolute path of the program executable in logs helps in precise identification of the executed program, which helps in tracing to pinpoint the location of the executable. Although the absolute paths of the program executables are available in the process control block (PCB), extraction requires dereferencing multiple layers of pointers in the kernel space, which degrades performance. Therefore, we use a separate eBPFmap data structure named `pid_exec_map`. The Host PID indexes this map and stores the executable file path as a value. The entries are added only when a new process is spawned (i.e., a `fork()/clone()` syscall invoked) and removed at the time of process termination (e.g., `exit()` syscall).

3) Host-Specific eBPFRing Buffer: For multiple microservices running on a host, each microservice produces its own set of application and system logs. Now, the logs generated by different microservices may be causally related, which XPLOG should capture. To ensure causal ordering among the inter-microservice logs, we maintain a host-specific eBPFring buffer to address the following two challenges.

1) Writing logs to a common buffer from multiple containers executing in different processor cores does not guarantee causal ordering, as the synchronization primitives may

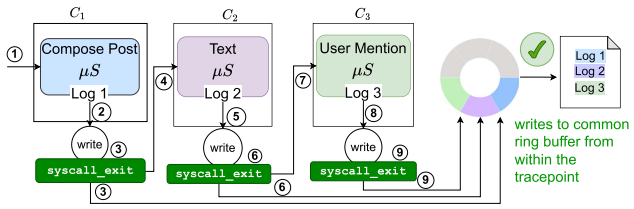


Fig. 2. Using eBPFring buffer to ensure causal ordering while generating a common log file.

enforce different ordering than the application event execution order, as discussed earlier. Therefore, the log messages need to be ordered at the time of execution of the underlying `write` syscall itself, as these are guaranteed to be successive for a given request pathway.

2) The significance of timestamps at the container level diminishes when comparing the log timestamps across containers due to the core-specific nature of timestamps obtained using `CLOCK_MONOTONIC`, lacking a global clock. Consequently, achieving total ordering through timestamps becomes unattainable.

To address these problems, a common log space is required which is accessible to a single source. A host-specific eBPFring buffer is suitable for this task. It serves as an event queue for both system and application logs, which are polled and consumed (Fig. 2). Consequently, the kernel-space component plays a crucial role in generating both system and application logs by collating them while preserving the causal order. To monitor application logs within the kernel space, specialized eBPFprobes are employed for the `write()` syscall. Whenever an application generates a log intended for storing in a specific log file, `write()` syscalls are triggered accordingly. By leveraging the file context information such as File Descriptors (FD), File Path, etc., obtained at the syscall entry probe, the system can determine the destination of the message string and ascertain whether it is a log-related operation, such as writing to `STDOUT/ STDERR` or a log repository (typically `/var/log/*`). If the `write` operation pertains to logging, the probe captures the message string and categorizes it as an application log. Consequently, `XPLOG` ensures the coherent aggregation of causally-ordered system and application logs. The generated log messages are periodically transmitted to the `XPLOG` *Collector* microservice.

Handling PID namespaces: A process operating within a container might have the same PID as another process running in a separate container. To distinguish logs originating from different containers, `XPLOG` incorporates a "*Task Context*" into the application logs (refer to Table II). This inclusion proves invaluable in segregating log contexts for processes across diverse containers. Consider a scenario where a microservice in Container 1 executes a request, followed by invoking another microservice in Container 2 (both residing on the same host). Both microservices generate application logs, and disambiguating these logs in a global file becomes challenging in the absence of a clear distinction. This difficulty arises due to the shared process namespace between containers and the host, with namespaces not visible per container. To address this issue, adding a "Task Context" (Host PID and Host TID) to the application logs clearly indicates which container generated a particular log. Furthermore, `XPLOG` uses an identifier, "*tag ID*" from microservice endpoints to filter application logs resulting from concurrent requests and corresponding system logs in the collated log message stream.

Atomic System Event Logging: `XPLOG` uses a novel approach for atomic system event logging to generate causally-consistent logs for the microservices running over a single host. A microservice, during its execution, typically triggers some system events in the kernel and generates log/debugging statements. These log/debugging statements are written to the log file by invoking the `write()` (or equivalent) syscall (Fig. 1 ①), which triggers a syscall entry tracepoint program (Fig. 1 ②) in the `XPLOG` Agent. This program populates the `pid_args_map` with the host PID & Thread ID and the syscall arguments (Fig. 1 ③). Then the system call executes (Fig. 1 ④), and once the execution is complete, it triggers the syscall exit tracepoint (Fig. 1 ⑤) to populate the executable file path of the event to the `pid_exec_map` (Fig. 1 ⑥). Consequently, the eBPFring buffer is populated with the log data by extracting the contents of the `pid_args_map` and `pid_exec_map` (Fig. 1 ⑦). Thus, for each pair of consecutive application events that are sequentially executed, all the associated log entries at the host OS kernel are registered together on the eBPFmaps, making them atomic records. The User-space component polls the ring buffer (Fig. 1 ⑧) and writes these atomic log entries to the `XPLOG` Collector Section IV-C (Fig. 1 ⑨). The steps between (Fig. 1 ②-⑤) help in writing the logs sequentially in the order of the corresponding event execution within the host OS kernel to create a unified causally-consistent log message stream from each individual hosts. Also, to preserve per-thread causal ordering on multi-core systems, `XPLOG` maintains a monotonic local logical counter [76] at the entry probe of each syscall. This counter is attached to the log entry and incremented per thread context. In practice, such a counter can be implemented in the host kernel using per-task data structures (e.g., `task_struct` in Linux) where a thread-local counter is updated atomically at each probe entry, thereby avoiding contention across cores. This counter is then used to sort logs before ingestion, ensuring intra-thread consistency despite potential out-of-order exits due to context switching. However, we still need to ensure causally-ordered logging across multiple hosts, as handled by the `XPLOG` Collector discussed next.

C. `XPLOG` *Collector*

The `XPLOG` Collector gathers the log entries transmitted by various `XPLOG` agents. To ensure causally-consistent logging across hosts with varying clocks, `XPLOG` employs "*Vector Clock*" [75] implemented in the kernel space triggered with each cross-host communication event. Fig. 3 demonstrates how `XPLOG` maintains causal consistency across a three-host microservice deployment. Each host has an initial vector clock: $H_1[1,0,0]$, $H_2[0,1,0]$, and $H_3[0,0,1]$. When a user writes a log message on Host 1 (ComposePost service), the local vector clock is incremented to $H_1[2,0,0]$, and `XPLOG` logs this event using the `write()` syscall. Later, when H_1 connects to H_2 (Text service) using connect(), `XPLOG` injects the current vector clock $H_1[2,0,0]$ into the syscall metadata. When this connection is accepted by H_2 using `accept()`, `XPLOG` updates

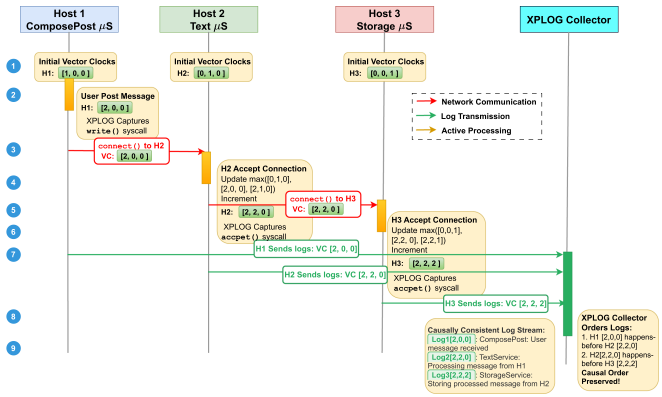


Fig. 3. Three hosts H_1(ComposePost), H_2(Text), H_3(Storage) processing a user request. Initial State: H1[1,0,0], H2[0,1,0], H3[0,0,1]. An example showing how vector clocks are updated during inter-host communication in a three-microservice deployment.

the vector clock of H_2 according to the merge rule: $VC[i] = \max(VC_{\text{local}}[i], VC_{\text{received}}[i]) \forall i$, resulting in $H_2[2,1,0]$ and then incrementing the local component to $H_2[2,2,0]$. This process recurs when H_2 connects to H_3 (Storage service), where H_3's vector clock is updated from [0,0,1] to [2,2,1] and then to [2,2,2] after incrementing the local component. Each syscall event creates a log entry with the current vector clock timestamp, which XPLOG agents send periodically to the collector. The collector applies the happens-before relation ((2)) to causally order the logs: $H_1[2,0,0] \rightarrow H_2[2,2,0] \rightarrow H_3[2,2,2]$ and makes the final log stream maintain the actual causal order of events on all hosts despite network latency or clock skew.

$$VC_1 < VC_2 \iff (\forall i.\ VC_1[i] \leq VC_2[i]) \wedge (\exists j.\ VC_1[j] < VC_2[j]) \quad (2)$$

In the vector clock implementation, we use a vector of integer values to represent the logical timestamp corresponding to each host. Application processes update the host OS's vector clock when an event occurs on the host (from any of the running microservices). During inter-host communication, such as when two microservices over two different hosts communicate, a network-related syscall is invoked. This syscall includes the vector clock of the host, which is stored in the eBPFmap. A host's local vector clock is updated based on the local events and the received timestamps over the network calls, following the standard vector clock update procedure [75]. Notably, we do not modify syscall payloads or application data, we only piggyback the vector clock into XPLOG's own logs. The collector utilizes this logical clock to consolidate logs from various hosts, ensuring causal ordering of events across the hosts.

V. Performance Evaluation

We evaluate XPLOG with the following objectives: (1) how well XPLOG can reduce disorders in the generated log compared to Tracee [33], a widely used system auditing framework that uses eBPFto capture runtime syscall execution logs, (2) runtime resource consumption overhead of XPLOG, (3) scalability, and (4) completeness and richness of the generated log information, compared to Tracee.

TABLE III
List of Monitored Syscalls

Syscall Type	List of Syscalls
File I/O	*READ, WRITE, OPEN, CLOSE, DUP, DUP2, DUP3, OPENAT, UNLINKAT*
Process	*CLONE, FORK, VFORK, EXECVE, EXIT, EXIT GROUP*
Socket	*CONNECT, ACCEPT, BIND, ACCEPT4, SEND, RECV, SOCKET*

A. Implementation Details

The source code, documentation, and the experimental configuration scripts of XPLOG implementation have been open-sourced[2]. The implementation of XPLOG Agent consists of 3,232 lines of code (LoC), while the XPLOG Collector is implemented with 176 LoC. During implementation, we have used C with libbpf library to realize eBPFprobe programs. We monitor 19 syscalls across file I/O, process, and socket types (Table III), which are configurable at runtime. We have developed corresponding eBPFprobes for each of them separately. Each syscall-generated log contains the fields listed in Table II. These fields are obtained using eBPFhelper functions, task struct, etc. As and when they are obtained, the fields are populated in the kernel-reserved space. Since the space reserved is a stream of bytes, it needs to be dereferenced appropriately depending on the syscall. Notably, each log entry spans between 500 to 600 bytes, encompassing comprehensive contextual information about a syscall, as generated by the eBPFinstrumentation framework. To identify the syscall, we have used the first 32 bits in the reserved space, which contain the syscall ID. Syscall logs and application logs are distinguished with the help of lms field setup in the log structure. To implement the Kernel-space Component, we have used the eBPFring buffer (described in Section IV-B3) size as 1 MB with a polling interval of 50 ms, which signifies that after each 50 ms, the logs stored in the ring buffer will be forwarded to the XPLOG collector. Additionally, we have reserved 2 MB for pid_exec_map and 64 KB for pid_args_map. The XPLOG Collector is implemented as a lightweight TCP server that receives serialized logs from agents over persistent sockets and immediately persists them in structured JSONL format with timestamps, host IDs, service names, and vector clock metadata.

B. Experimental Setup

To evaluate the efficacy of XPLOG, we use DeathStarBench [32], an open-source benchmark suite for cloud microservices. We have used the *Social Network* microservice of DeathStarBench for its high service composition (+30 microservices) and rich interaction patterns for large-scale testing of microservice applications. This particular application is composed of 30 different microservices with 3 different active endpoints for user interaction, such as Compose-Post-service (CP), User-Timeline-service (UT) and Home-Timeline-service (HT) where each client request directly impacts

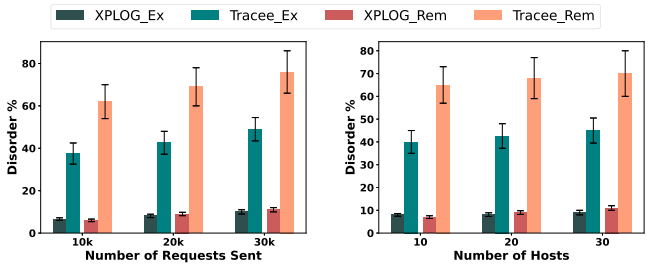
[2]https://github.com/usatpath01/Pluggable_Logging (Accessed: 2025/10/03 10:50:39)

different microservices; primarily (a) "*Load balancer*", (b) "*Compose Post*", (c) "*Text*", and (d) "*User Mention*". However, these microservices make use of several other microservices and trigger them as and when required[3] (like, "*post storage*", "*user timeline*","*Rabbit MQ*", etc.) during the processing of individual requests. We have used an HTTP benchmarking tool for workload generation, called `wrk` [4], which sends a variable number of multithreaded asynchronous requests ranging from 10,000 to 30,000 parallel requests targeted towards the services and look into the consolidated log generated by the application. To test the multi-host log collection capability of `XPLOG`, we have deployed the microservices in 10-30 VMs, each configured as edge computing hosts running multiple containerized microservices using Docker Swarm as container orchestration. Each VM is configured with Ubuntu 22.04 SMP with Linux kernel version $6.5.0-41-generic$, each with 16 vCPU cores and $64\,GB$ of RAM. Among these hosts, one is assigned as a Docker Swarm manager and the rest as workers. The Docker manager host schedules the microservices across the worker hosts. To prevent underutilization of the VMs, we have replicated the services with a replication factor ranging from 3 to 5; consequently, a minimum of 4 to 5 microservices are scheduled per host machine. We have used Docker Swarm as an orchestrator due to its simpler multi-host orchestration and lightweight configuration. Notably, `XPLOG` can also be deployed as a DaemonSet in Kubernetes, enabling per-node `XPLOG` Agents to monitor all pods scheduled on that node.

For comparison, we collect logs using (a) `Tracee` [33] and (b) the proposed `XPLOG`. Notably, `Tracee` also uses eBPFto capture the runtime system events and log them based on the timestamped order. The `XPLOG` Agents are deployed as separate containers in each worker, and the `XPLOG` Collector in the manager host. As `Tracee` does not support multi-host log collection, we deploy `Tracee` in each host, and log message streams are collected using `SFTP`, merged, and sorted based on the timestamp available for each log entry.

C. Analysis of Causal Ordering Level

Ground truth: To show the level of causally ordered logs for multiple parallel microservices, we have used real-time sequential logs as the ground-truth baseline by sending sequential requests to the application endpoints (CP, UT, and HT) on a single host machine while ensuring synchronized `CLOCK_MONOTONIC` across all the processor cores for that host. Notably, the log generated through sequential requests on such a single host machine will always maintain a true causal order without any interference from parallel requests. For comparison, we collected `XPLOG`-generated and `Tracee`-generated logs while sending 10,000–30,000 parallel and asynchronous requests to the target application over multiple hosts, without any explicit clock synchronization. These requests were issued over a 120-second interval, corresponding to an average load of ∼80 to ∼250 requests per second. To emulate real-world user interaction patterns, we also varied the request rates and service endpoints randomly during execution. We have also collected the logs by randomly varying the requests and service endpoints to replicate a real-world user interaction.



(a) #Requests vs Log Disorder (#Hosts: 20)

(b) #Hosts vs Log Disorder (Concurrent Req: 20K)

Fig. 4. Log disorder between `Tracee` and `XPLOG` (Service endpoint: CP+UT+HT)

TABLE IV
IRRELEVANT LOGS RECEIVED VS RELEVANT LOGS LOST

Events	Ground Truth Logs Count	Tracee			XPLOG		
		Irrelevant Logs Received	Relevant Logs Lost	Query Time (sec)	Irrelevant Logs Received	Relevant Logs Lost	Query Time (sec)
e_1: Triggered by CP	83	2586	28	3.015	0	0	1.201
e_2: Triggered by UT	85	2622	40	3.019	0	0	1.233
e_3: Triggered by HT	92	2608	35	3.023	0	0	1.305

To measure the causal distance between the baseline (sequentially-ordered) log and the log generated by the two frameworks, we use two different disorder metrics adopted from the existing literature [77]: (a) `exchange (Ex) %`: which measures the minimum number of entries that require swapping to order a sub-sequence concerning the baseline, with respect to total number of log lines and (b) `rem%`: which measures the minimum number of entries that must be removed to order a sub-sequence concerning the baseline with respect to total number of log lines.

Fig. 4(a) shows the percentage of disorder in terms of `Ex` and `rem` with respect to the number of concurrent requests from the clients. We observe that the measure of disorder is significantly lower in `XPLOG` compared with `Tracee`, proving that `XPLOG`-generated logs are closer to the baseline (sequentially-ordered log) and thus closer to true causal ordering. As shown in Fig. 4(a), for a service endpoint: CP+UT+HT, `XPLOG` generates only 83 log lines, among which 2 to 3 log lines require swapping (EX) and 2 to 3 log lines require removal (REM) to match the baseline ordering. In contrast, `Tracee` records 2,586 log lines, of which 1,034 require swapping and 1,552 need removal to match the same baseline. This significant reduction in disorder is due to `XPLOG`'s selective filtering of irrelevant system calls and its causality-preserving design. Notably, for 30,000 parallel requests for the same set of microservices, `Tracee` generated almost 3.5 GB and comparatively `XPLOG` generates 5.7 GB of log messages in total (more details in Section V-E, Table V). Similarly, Fig. 4(b) shows the disorder percentage with respect to the increasing number of host machines. We observe that `Ex` and `rem` percentages are $6x$-$8x$ less in `XPLOG` compared to `Tracee`, even when the number of hosts increases. The negligible disorder shown by `XPLOG` is due to the bufferbloat at the communication channel

[3]https://github.com/delimitrou/DeathStarBench/raw/master/socialNetwork/figures/socialNet_arch.png (All URLs in this paper have accessed last on 2025/10/03 10:50:39)

[4]https://github.com/wg/wrk (Accessed: 2025/10/03 10:50:39)

TABLE V
LOG SIZES VS #REQUESTS (SERVICE ENDPOINT: CP)

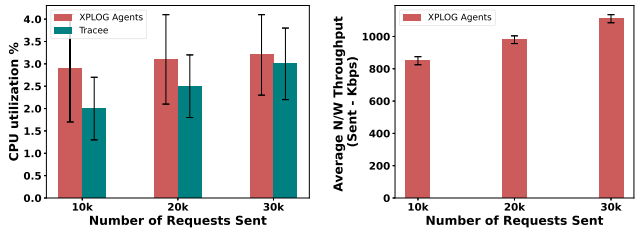
Concurrent Requests Sent	Total Log Size (≈ GB)		Increase In Log Content (times)
	Tracee	XPLOG	
10,000	2.1	5.7	2.7x
20,000	3.5	11	3.1x
30,000	5.2	13	2.5x

between the agents and the collectors when there is a spike in the number of generated log messages; however, we observe that the host-generated logs are truly causal for $> 99\%$ of the experiment scenarios. While XPLOG significantly improves causal consistency in logging, it currently only supports Linux-based environments.

D. Analysis of Runtime Observability

To test the capability of XPLOG in producing relevant logs with minimal query processing time, we apply an event-sequence-based filter on the generated log streams to extract the necessary information. The results are summarized in Table IV. The table shows the number of "relevant logs lost" and "irrelevant logs received", as extracted from the log streams for an event when applying the filter. We define irrelevant logs as those captured by the logging framework that do not pertain to the specific event or context being investigated, as they include data from unrelated processes or activities, making it harder to focus on the required information. In contrast, relevant logs are pertinent to the specific event or context but were not captured by the logging tool. Missing these logs can lead to incomplete information, potentially hindering effective monitoring, debugging, and investigation.

In Table IV, we present the results for relevant logs lost, irrelevant logs received, and the time taken to extract query results for three different types of events observed by the system administrator. The events are e_1: triggered by the CP endpoint, e_2: triggered by the UT endpoint, and e_3: triggered by the HT endpoint. The timestamp of the event serves as the input. We ran both Tracee and XPLOG alongside the microservices to demonstrate our findings. XPLOG is specifically designed to use tag IDs and XPLOG IDs to filter logs. To establish the ground truth, we repeatedly sent the requests to the 3 endpoints to determine the number of logs generated for that specific event. This approach helps understand the typical volume of logs and their variability, which ranges between 80 to 100 log messages per event due to context switches in the system during the event processing. The table shows that the query processing time for Tracee is almost $3x$ that of XPLOG to figure out the relevant logs from the collated log message stream. However, it returns many irrelevant logs (as the log messages are disordered) while missing several of the relevant log messages. In contrast, XPLOG can filter out the exact log messages corresponding to an event following its capability of preserving the causal order of the log messages.



(a) Avg. CPU Utilization (b) Avg. Netw. Traffic to Collector

Fig. 5. Average CPU (both kernel-space overhead from eBPF probes and user-space overhead from XPLOG agent log processing and transmission) and network utilization of 19 XPLOG agents.

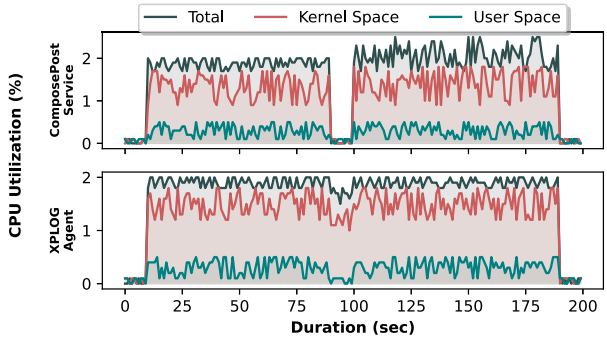


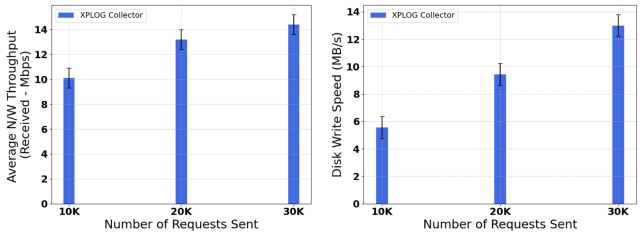
Fig. 6. CPU utilization per request (Service endpoint: CP).

E. Resource Overhead Analysis

To measure the overhead of XPLOG, we have used cAdvisor [5], an open-source tool developed by Google to monitor containers. Also, to measure the resource consumption by the hosts, we have used NodeExporter [6] to measure the CPU, Memory, and Network utilization of each host.

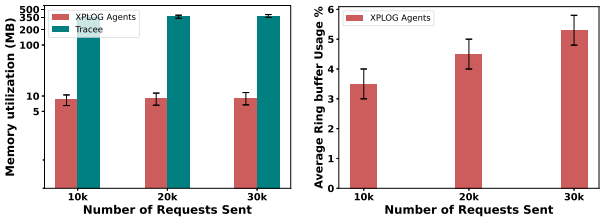
1) CPU and Network Utilization for XPLOG Agents: Fig. 5 shows the total CPU (kernel-space and user-space) and network utilization of 19 XPLOG agents running in the workers. We observe that the average CPU usage (see Fig. 5(a)) of XPLOG Agents are 2.9%, 3.10%, and 3.15% with error bars showing a variance of about $\pm 0.5 - 0.8\%$. when 10 K, 20 K, and 30 K requests are sent concurrently. This indicates that even under bursty request patterns, CPU overhead remains stable and low, without significant deviation across runs. The overhead measurements show the total CPU utilization observed in the container, including system call latency from kernel probes and the resource usage of XPLOG's user-space agent. In Fig. 6, we show the % CPU utilization in terms of kernel space and user-space of XPLOG Agent and application service. For the XPLOG Agents, the majority of CPU utilization is due to the underlying eBPFprograms; thus, utilization by kernel-space processes is comparatively higher than user-space processes (shown as the red shaded area) at the application services. These results show that the proposed XPLOG framework is a low-overhead solution with minimal processing requirements. We also show the network utilization (see Fig. 5(b)) of the XPLOG agents,

[5]https://github.com/google/cadvisor (Accessed: 2025/10/03 10:50:39)
[6]https://github.com/prometheus/node_exporter (Accessed: 2025/10/03 10:50:39)



(a) Avg. N/W Throughput (b) Avg. Disk Write Speed

Fig. 7. Average network utilization and disk write speed of XPLOG collector.



(a) Avg. Memory Usage (b) eBPF Buffer Usage

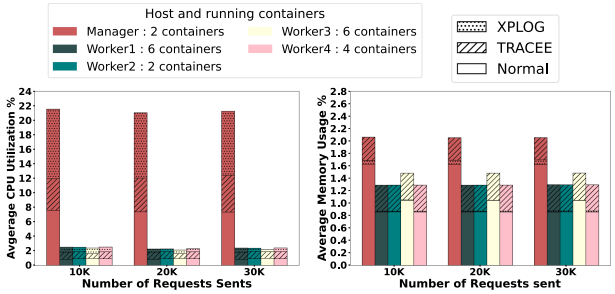
Fig. 8. Average memory consumption and eBPF ring buffer utilization of 19 XPLOG Agents for workloads of 10 K, 20 K, and 30 K requests within a 120-second interval.

which is $\sim$ 1.7 Mbps, because of the constant streaming of the generated log messages to the collector.

2) Resource Usage by XPLOG *Collector:* Fig. 7(a) shows the the XPLOG collector's network throughput, which sustained progressively higher ingestion rates as the workload increased, starting from 10.3 Mbps at a load of 10 K requests, rising to 13.2 Mbps at 20 K requests, and reaching 14.3 Mbps at 30 K requests. On the other hand, Fig. 7(b) shows the disk write performance of XPLOG collector. At 10 K requests, the Collector achieved a sustained write speed of 5.2 MB/s, which increased to 9.0 MB/s at 20 K requests and further to 12.8 MB/s at 30 K requests.

3) Memory Overhead: We observe that the memory overhead (see Fig. 8(a)) of XPLOG Agents remains constant with 10 K, 20 K, and 30 K requests being sent, whereas the memory usage by the XPLOG Collector varies from 2.21 MB to 2.7 MB. To measure the size of the total log messages generated by XPLOG, we observed that the logging framework generates 5.7 GB when the number of parallel requests is 10,000 whereas Tracee generates 2.1 GB of log messages. Notably, this total size of the log messages increased to 5.2 GB and 13 GB for 30,000 concurrent requests for Tracee and XPLOG, respectively (see Table V). Notably, XPLOG also incorporates the application logs generated from the microservices within the global log, whereas Tracee only captures the syscall logs; therefore, the number of log entries in the XPLOG-generated log is comparatively higher than Tracee-generated logs, as indicated in Table V.

XPLOG can handle this high amount of logs reasonably well, which can be justified by Fig. 8(a) where the memory requirement of the agents does not increase significantly even in case of a high request load. We also observe that (see Table IV), a significant amount (i.e., $\approx$ 2/3rd of all logs) of the generated logs are due to the sandboxing environment and platform. Rather



(a) Avg. CPU Utilization of Hosts on different loads (b) Avg. Memory Utilization of Hosts on different loads

Fig. 9. Host Resource utilization (Service endpoint: CP+UT+HT).

than disposing of these logs, XPLOG uses a different mechanism to separate them (XPLOG -relevant logs, as indicated in Section V-D), resulting in meaningful log entries in the collated log message stream.

The percentage utilization of the eBPFring buffer while varying the number of requests from 10 K to 30 K is presented in Fig. 8(b). We have declared the eBPF ring buffer with `__uint(max_entries, 256 * 1024)`, which sets its total capacity to 262,144 bytes ($\approx$ 256 KB). Given that each log entry in our implementation (Refer Listing 1) is approximately 500 to 600 bytes, the buffer can store up to around 437 entries before reaching full capacity. Since the buffer is polled every 50 ms, this translates to a sustained handling capability of roughly 8,740 log entries per second without overflow. We observe that the peak utilization of eBPFring buffer is 5.2% for CP, UT, and HT endpoints.

Notably, the bounded in-kernel ring buffer may limit performance, as once full it overwrites older entries, causing potential log loss during short high-throughput bursts. Hence, buffer size must be chosen carefully by administrators based on application log rates. Nevertheless, XPLOG's vector clock–based timestamping preserves causal relationships among remaining events, ensuring logs across threads and hosts can be reconstructed in correct partial order. Thus, while minor detail loss may occur, the system avoids analysis blind spots from event reordering or omission.

4) Host-Level Resource Utilization: We have also measured the resource utilization of hosts to determine how the overall system handles increasing loads (shown in Fig. 9). For this load testing, we follow constant duration with a variable request rate strategy where we vary the number of requests, keeping the duration of the test constant at 120 seconds, i.e., for 10,000 requests, the rate of request is 83; for 20,000, it is 166, and for 30,000, it is 250 requests per second. We observe that the CPU Utilization of the manager is around 7% when only the microservices are running, whereas, with Tracee, it is 12% (shown in the hashed line of Fig. 9(a)). However, with XPLOG, it goes up to 21% (shown in dotted of Fig. 9) as the collector continuously consumes the log messages sent by the agents. However, we observe that the memory usage (shown in Fig. 9(b)) by the hosts when only the microservices are running is 1.6%; with XPLOG, it is 1.7%, and with Tracee, it is 2.3%. This is because the XPLOG agent sends the logs to the collector service where Tracee stores them in the host itself. These results indicate that while XPLOG introduces a marginally higher CPU

TABLE VI
LOG WRITING LATENCY (#CONCURRENT REQUESTS: 10,000)

Type of Requests	Log entries (≈) per Timestamp	Log Size (MB)	Peak Latency (Sec)
CP	3.1K	1.9	1.209
CP & UT	3.6K	2.2	1.238
CP, UT & HT	3.9K	2.3	1.318

overhead due to the continuous log transmission to the collector service, it achieves better memory efficiency by offloading log storage. This trade-off can be advantageous in scenarios for memory-intensive application workloads, which is typical for edge Computing-on-Demand(CoD) scenarios [78], [79]. The overall lower memory footprint of XPLOG makes it a more viable option for environments with high memory usage demands, highlighting its suitability for scalable and efficient logging in distributed systems. Notably, reduced resource consumption also lowers operational costs, particularly in cloud environments where resources are billed based on usage.

To demonstrate the real-time collection of logs, which depends on the underlying network latency, we have used the Linux `tc` utility to configure latency between each host and collector as 0.5 ms. For 10,000 parallel requests on different application endpoints, we have calculated the latency between the generation of a log message and the appearance of the same message at the XPLOG collector. The results (see Table VI) show that the average latency is close to 1 s, indicating that the proposed observability framework makes the log messages available to the downstream applications reasonably fast.

F. Qualitative Evaluation

As part of our qualitative analysis, we focus on two primary factors: (a) ease of log consultation and (b) richness level of logs. We have used a sample application log and system log entry format in Listing 1 with relevant fields highlighted for ease of understanding[7].

1) Ease of Log Consultation: As shown in Table V, XPLOG generates much more logs than the existing systems like Tracee, while combining the application logs with the system logs. One pertinent question is how difficult it is to identify contextual information from the generated logs. To determine the contextual information, a system administrator tries to isolate the logs relevant to a particular request during log analysis and then looks into the sequence of events that happened while processing that request. To extract the application logs from the entire log file, she can filter the log file with the request Identifier ("*tag ID*"), which is available inside the `lms` field (see Line 1 of Listing 1). This process will provide all the application logs generated by all the microservices across all the hosts. The `host_pid` and `host_tid` fields together can be used to identify all the logs generated by all the microservices in each host during the processing of the request as shown in Listing 1 (see Lines 13, 21, 22, 23). Although it may look a bit complicated, a simple script[8]

```
1  /* Application Log */
2  Host 1 : {
3      "event_context": {
4          "ts": XXX,"datetime": "XXX",
5          "task_context": {
6              "host_pid": 314463, "host_tid": 317863,
7              "task_command" : "ComposePostServ" ··· }},
8      "data": {
9          "lms": "[2024-Jul-26 15:33:55.832728] <info>: (
       ComposePostHandler.h:370: ComposePost) Request
       Order : 1, ID : 899493982365583360"},
10     "artifacts": {"exe": "/usr/local/bin/
       ComposePostService"}}
11 Host 2 : { ··· "task_context" : { "host_pid" :  1532108,
        "host_tid" :  1535513, ··· }, "data" : { "lms" :
       ["2024-Jul-26 15:33:55.844358] <info>: (TextHandler
       .h:46:ComposeText) Request Order : 2 , ID :
       899493982365583360" }, "artifacts" : { "exe" : "/
       usr/local/bin/TextService" } },
12 /* System Log */
13 Host 1: {
14     "event_context": {
15         "ts": XXX, "datetime": "XXX",
16         "syscall_id": 42,"syscall_name": "connect",
17         "task_context": {
18         "host_pid": 314463, "host_tid": 317863, ···}},
19     "arguments": { "uservaddr":"0x80003760","addrlen
       ":16},
20     "artifacts": {"exe": "/usr/local/bin/
       ComposePostService","IP":"10.11.0.63","port
       ":"33315"}}
21 Host 1 : { ··· "syscall_name" : "read", "task_context" :
        { "host_pid" :  314463, "host_tid" :  317863, ···
       "artifacts" : { "exe" : "/usr/local/bin/
       ComposePostService", "file_read" : "/var/lib/docker
       /containers/···/resolv.conf" }}},  ···
22 Host 1 : { ··· "syscall_name" : "send",  "task_context"
       : { "host_pid" :  314463, "host_tid" :  317863,
       ··· }}
23 Host 2 : { ··· "syscall_name" : "recv",  "task_context"
       : { "host_pid" :  1532108, "host_tid" :  1535513,
       ··· }},
```

Listing 1: Example of XPLOG-generated log entries

can be used to realize the same. From the real example logs (shown partially for brevity), it is visible that both application and system logs from multiple systems can be accessed as per their causal ordering from the output (see Listing 1).

2) Log-Richness Level: As both `Tracee` and XPLOG are eBPF-based approaches, they can monitor selected system calls, but in terms of expressibility, both frameworks differ significantly. To compare this feature, we have customized both frameworks to monitor a list of system calls as listed in Table III and collected the logs while executing the application above using both the frameworks.

We observe that XPLOG-generated application and system logs (Listing 1) provide more information in comparison to the `Tracee`-generated logs (Listing 2). A summary of the comparison (Table VII) shows that `Tracee` cannot provide application logs. Further, understanding the context is complicated as there are overlapping logs (i.e., without causal order, see `TIME` field of Line 4 in Listing 2). Moreover, `Tracee` provides only the file descriptors value, where XPLOG provides more readable path information (see Line 21 of Listing 1). Additionally, XPLOG captures the syscall arguments, which provide several additional information, like the network addresses from socket calls (see Line 22 of Listing 1), contents read/written to a file from the syscall arguments fields, etc. Therefore, compared to `Tracee`,

[7]Each log entry contains fields as described in Table II. Sample logs for XPLOG and Tracee are available at https://github.com/usatpath01/Pluggable_Logging/tree/main/Logs (Accessed: 2025/10/03 10:50:39)

[8]https://github.com/usatpath01/Pluggable_Logging/tree/main/scripts (Accessed: 2025/10/03 10:50:39)

TABLE VII
COMPARISON OF CAPABILITIES BETWEEN Tracee AND XPLOG

	Tracee	XPLOG
Task Context	✓	✓
Return Values	✓	✓
Container Isolation	✓	✓
Application Logs	✗	✓
Executable Path	✗	✓
File Names, Socket Address	✗	✓
Request Identification	✗	✓
Multi-Host Support	✗	✓

```
1 TIME    UID  COMM  PID  TID RET EVENT   ARGS
2 ...
3 21:20:32:498568 0   PostStorageServ 1   1   0
      security_socket_bind    sockfd: 10, local_addr: map[
      XXX:0.0.0.0 sin_port:9090] 21:20:27:055062  0
      containerd-shim  798120  798120  0  sched_process_exec
              cmdpath: /usr/bin/containerd-shim-runc-v2,
      pathname: /usr/bin/containerd-shim-runc-v2, dev:
      8388611, inode: 3285194, ctime: 1704856244464194144,
      inode_mode: 33261, interpreter_pathname: <nil>,
      interpreter_dev: <nil>, interpreter_inode: <nil>,
      interpreter_ctime: <nil>, argv: [/usr/bin/containerd-
      shim-runc-v2 -namespace moby -address /run/containerd/
      containerd.sock], invoked_from_kernel: 0, env: <nil>
4 21:20:34:435046  0      ComposePostServ  1       1       0
            security_socket_accept    sockfd: 8, local_addr
      : map[XXX:0.0.0.0 sin_port:9090]
5 21:20:34:448360  0      TextService      1       1       0
            security_socket_accept    sockfd: 8, local_addr
      : map[XXX:0.0.0.0 sin_port:9090]
6 ...
```

Listing 2: Tracee-generated syscall log snippet

XPLOG is easy to understand and learn for sysadmins due to its expressiveness.

However, there are several avenues to further enhance the robustness and efficacy of XPLOG. Although XPLOG supports both system- and application-level logging, its current evaluation primarily focuses on low-level audit tools such as Tracee. It does not yet integrate with full-stack observability platforms like Pixie[9], which operate at the RPC tracing and telemetry layers. As a promising future direction, we plan to extend XPLOG to export causally tagged system and application logs to OpenTelemetry-compatible pipelines and integrate with tools like Pixie. This would broaden XPLOG's applicability within modern observability ecosystems and enable more comprehensive end-to-end visibility.

VI. CONCLUSION

This paper introduced an innovative and non-invasive platform observability framework called XPLOG that leverages eBPFto generate comprehensive logs for distributed edge applications. Unlike prior frameworks such as KubeArmor[10] that primarily focus on security policy enforcement and audit logging at the container boundary without integrating application-level events or supporting unified provenance reconstruction across distributed execution paths, XPLOG is specifically designed to enable comprehensive, provenance-aware observability. The framework effectively captures both application and system logs, maintaining causal consistency to provide a holistic view of events. Deployed as a lightweight memory-efficient module, XPLOG reduces the disorder among the generated log entries while preserving the causal ordering of logs with respect to the sequential execution of corresponding microservices. Notably, XPLOG enables detailed extraction of datapath executions from parallel microservices, including valuable information like binary executable file paths and accessed files, a unique feature absent in the existing logging frameworks. The resulting logs are significantly helpful for downstream services that need runtime system observability.

XPLOG can be further enhanced through efficiency-oriented optimizations such as coalescing. At present, it does not support hosts with multiple operating systems, making it unsuitable for heterogeneous platforms. A promising extension is to export tagged logs into OpenTelemetry pipelines and integrate with frameworks like Pixie, enabling broader observability and improved end-to-end visibility. Overall, XPLOG offers a strong foundation for causality-preserving observability in distributed microservices and opens directions for research in hybrid log integration, cross-platform support, and automated provenance-based analytics in cloud-edge deployments.

[9]https://px.dev/ , Accessed: 2025/10/03 10:50:39
[10]https://kubearmor.io/ , Accessed: 2025/10/03 10:50:39

ACKNOWLEDGMENT

Neha Dalmia and Rajat Bachhawat were associated with the Indian Institute of Technology Kharagpur during this work.

REFERENCES

[1] W. U. Hassan, M. A. Noureddine, P. Datta, and A. Bates, "OmegaLog: High-fidelity attack investigation via transparent multi-layer log analysis," in *Proc. 27th Netw. Distrib. Syst. Secur.*, 2020.
[2] P. Datta, I. Polinsky, M. A. Inam, A. Bates, and W. Enck, "ALASTOR: Reconstructing the provenance of serverless intrusions," in *Proc. 31st USENIX Secur.*, 2022, pp. 2443–2460.
[3] "Sysdig - the cloud moves fast. your security should too," 2013. Accessed: Jul. 24, 2025. [Online]. Available: https://sysdig.com/
[4] T. Taylor, F. Araujo, and X. Shu, "Towards an open format for scalable system telemetry," in *Proc. IEEE Int. Conf. Big Data*, 2020, pp. 1031–1040.
[5] R. Sekar, H. Kimm, and R. Aich, "eAudit: A fast, scalable and deployable audit data collection system," in *Proc. 45th IEEE Symp. Secur. Privacy*, 2024, pp. 3571–3589.
[6] W. Kellerer, P. Kalmbach, A. Blenk, A. Basta, M. Reisslein, and S. Schmid, "Adaptable and data-driven softwarized networks: Review, opportunities, and challenges," *Proc. IEEE*, vol. 107, no. 4, pp. 711–731, Apr. 2019.
[7] M. Scrocca, R. Tommasini, A. Margara, E. D. Valle, and S. Sakr, "The Kaiju project: Enabling event-driven observability," in *Proc. 14th ACM Int. Conf. Distrib. Event-Based Syst.*, 2020, pp. 85–96.
[8] S. Zawoad, R. Hasan, and K. Islam, "SECProv: Trustworthy and efficient provenance management in the cloud," in *Proc. IEEE Conf. Comput. Commun.*, 2018, pp. 1241–1249.
[9] T. Blount, A. Chapman, M. Johnson, and B. Ludascher, "Observed vs. possible provenance (research track)," in *Proc. 13th Int. Workshop Theory Pract. Provenance*, 2021.
[10] M. Zipperle, F. Gottwalt, E. Chang, and T. Dillon, "Provenance-based intrusion detection systems: A survey," *ACM Comput. Surv.*, vol. 55, no. 7, pp. 1–36, 2022.
[11] M. M. Anjum, S. Iqbal, and B. Hamelin, "ANUBIS: A provenance graph-based framework for advanced persistent threat detection," in *Proc. 37th ACM/SIGAPP Symp. Appl. Comput.*, 2022, pp. 1684–1693.
[12] A. Chen, Y. Wu, A. Haeberlen, W. Zhou, and B. T. Loo, "Differential provenance: Better network diagnostics with reference events," in *Proc. ACM Workshop Hot Topics Netw.*, 2015, pp. 1–7.

[13] A. Tabiban, H. Zhao, Y. Jarraya, M. Pourzandi, M. Zhang, and L. Wang, "ProvTalk: Towards interpretable multi-level provenance analysis in networking functions virtualization (NFV)," in *Proc. 29th Netw. Distrib. Syst. Secur.*, 2022.

[14] Y. Gan, G. Liu, X. Zhang, Q. Zhou, J. Wu, and J. Jiang, "Sleuth: A trace-based root cause analysis system for large-scale microservices with graph neural networks," in *Proc. 28th ACM Int. Conf. Architectural Support Program. Languages Operating Syst.*, 2023, pp. 324–337.

[15] A. R. Bernat and B. P. Miller, "Anywhere, any-time binary instrumentation," in *Proc. 10th ACM SIGPLAN-SIGSOFT Workshop Prog. Anal. Softw. Tools*, 2011, pp. 9–16.

[16] F. Wang and Y. Shoshitaishvili, "Angr-the next generation of binary analysis," in *Proc. IEEE Cybersecur. Develop.*, 2017, pp. 8–9.

[17] Y. Shoshitaishvili et al., "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *Proc. IEEE Symp. Secur. Privacy*, 2016, pp. 138–157.

[18] Elastic, "Logstash: Collect, parse, transform logs| Elastic," 2010. Accessed: Feb. 4, 2025. [Online]. Available: https://www.elastic.co/logstash/

[19] Fluent, "Fluentd | open source data collector | unified logging layer," Jun. 2024, Accessed: Feb. 4, 2025. [Online]. Available: https://www.fluentd.org/

[20] X. Merino and C. E. Otero, "The cost of virtualizing time in linux containers," in *Proc. 8th IEEE Cloud Summit*, 2022, pp. 63–68.

[21] M. Cinque, R. D. Corte, and A. Pecchia, "Microservices monitoring with event logs and black box execution tracing," *IEEE Trans. Serv. Comput.*, vol. 15, no. 1, pp. 294–307, Jan./Feb. 2022.

[22] X. Han, T. Pasquier, A. Bates, J. Mickens, and M. Seltzer, "Unicorn: Runtime provenance-based detector for advanced persistent threats," in *Proc. 27th Annu. Netw. Distrib. Syst. Secur. Symp.*, San Diego, CA, USA, 2020.

[23] A. Goyal, X. Han, G. Wang, and A. Bates, "Sometimes, you aren't what you do: Mimicry attacks against provenance graph host intrusion detection systems," in *Proc. 30th Annu. Netw. Distrib. Syst. Secur. Symp.*, San Diego, CA, USA, 2023.

[24] M. A. Vieira, M. S. Castanho, R. D. Pacífico, E. R. Santos, E. P. C. Júnior, and L. F. Vieira, "Fast packet processing with eBPF and XDP: Concepts, code, challenges, and applications," *ACM Comput. Surv.*, vol. 53, no. 1, pp. 1–36, 2020.

[25] L. Security, "Use of eBPF as a more secure alternative to kernel level observability," Jan. 2022, Accessed: Feb. 4, 2025. [Online]. Available: https://redcanary.com/blog/ebpf-for-security/

[26] B. Gregg, "Learning eBPF and BCC," Jan. 2019, Accessed: Feb. 4, 2025. [Online]. Available: https://www.brendangregg.com/blog/2019-01-01/learn-ebpf-tracing.html

[27] T. L. Foundation, "The state of eBPF," Jan. 2024, Accessed: Feb. 4, 2025. [Online]. Available: https://www.linuxfoundation.org/hubfs/eBPF/The_State_of_eBPF.pdf

[28] F. Neves, N. Machado, R. Vilaça, and J. Pereira, "Horus: Non-intrusive causal analysis of distributed systems logs," in *Proc. 51st Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2021, pp. 212–223.

[29] C. Liu, Z. Cai, B. Wang, Z. Tang, and J. Liu, "A protocol-independent container network observability analysis system based on eBPF," in *Proc. IEEE 26th Int. Conf. Parallel Distrib. Syst.*, 2020, pp. 697–702.

[30] J. Levin and T. A. Benson, "ViperProbe: Rethinking microservice observability with eBPF," in *Proc. IEEE 9th Int. Conf. Cloud Netw.*, 2020, pp. 1–8.

[31] F. Neves et al., "Falcon: A practical log-based analysis tool for distributed systems," in *Proc. 48th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2018, pp. 534–541.

[32] Y. Gan et al., "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proc. 24th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2019, pp. 3–18.

[33] A. Security, "Tracee| Github," Jun. 2024, Accessed: Feb. 4, 2025. [Online]. Available: https://github.com/aquasecurity/tracee

[34] M. A. Inam et al., "SoK: History is a vast early warning system: Auditing the provenance of system intrusions," in *Proc. IEEE Symp. Secur. Privacy*, 2023, pp. 2620–2638.

[35] D. Huye, Y. Shkuro, and R. R. Sambasivan, "Lifting the veil on Meta's microservice architecture: Analyses of topology and request workflows," in *Proc. USENIX Annu. Tech. Conf.*, 2023, pp. 419–432.

[36] B. Debnath et al., "LogLens: A real-time log analysis system," in *Proc. IEEE 38th Int. Conf. Distrib. Comput. Syst.*, 2018, pp. 1052–1062.

[37] K. Suo, Y. Zhao, W. Chen, and J. Rao, "vNetTracer: Efficient and programmable packet tracing in virtualized networks," in *Proc. 38th IEEE Int. Conf. Distrib. Comput. Syst.*, 2018, pp. 165–175.

[38] X. Ge, B. Niu, and W. Cui, "Reverse debugging of kernel failures in deployed systems," in *Proc. USENIX Annu. Tech. Conf.*, 2020, pp. 281–292.

[39] S. Y. Lim, B. Stelea, X. Han, and T. Pasquier, "Secure namespaced kernel audit for containers," in *Proc. 12th ACM Symp. Cloud Comput.*, 2021, pp. 518–532.

[40] J. Zeng, Z. L. Chua, Y. Chen, K. Ji, Z. Liang, and J. Mao, "WATSON: Abstracting behaviors from audit logs via aggregation of contextual semantics," in *Proc. Netw. Distrib. Syst. Secur.*, 2021.

[41] U. Satpathy, R. Thakur, S. Chattopadhyay, and S. Chakraborty, "DisProTrack: Distributed provenance tracking over serverless applications," in *Proc. IEEE Conf. Comput. Commun.*, 2023, pp. 1–10.

[42] L. Zhou et al., "Hardware-assisted live kernel function updating on intel platforms," *IEEE Trans. Dependable Secure Comput.*, vol. 21, no. 4, pp. 2085–2098, Jul./Aug. 2024.

[43] M. N. Hossain et al., "Sleuth: Real-time attack scenario reconstruction from cots audit data," in *Proc. 26th USENIX Secur. Symp.*, 2017, pp. 487–504.

[44] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proc. 22nd ACM Symp. Operating Syst. Princ.*, 2009, pp. 117–132.

[45] K. Kc and X. Gu, "ELT: Efficient log-based troubleshooting system for cloud computing infrastructures," in *Proc. 30th IEEE Int. Symp. Reliable Distrib. Syst.*, 2011, pp. 11–20.

[46] F. Yang, J. Xu, C. Xiong, Z. Li, and K. Zhang, "PROGRAPHER: An anomaly detection system based on provenance graph embedding," in *Proc. 32nd USENIX Secur. Symp.*, 2023, pp. 4355–4372.

[47] M. Stamatogiannakis, E. Athanasopoulos, H. Bos, and P. Groth, "Prov 2r: Practical provenance analysis of unstructured processes," *ACM Trans. Internet Technol.*, vol. 17, no. 4, pp. 1–24, 2017.

[48] M. Backes, S. Bugiel, and S. Gerling, "Scippa: System-centric IPC provenance on android," in *Proc. 30th ACM Annu. Comput. Secur. Appl. Conf.*, 2014, pp. 36–45.

[49] D. J. Pohly, S. McLaughlin, P. McDaniel, and K. Butler, "Hi-Fi: Collecting high-fidelity whole-system provenance," in *Proc. 28th ACM Annu. Comput. Secur. Appl. Conf.*, 2012, pp. 259–268.

[50] C. Collberg, A. Gibson, S. Martin, N. Shinde, A. Herzberg, and H. Shulman, "Provenance of exposure: Identifying sources of leaked documents," in *Proc. IEEE Conf. Commun. Netw. Secur.*, 2013, pp. 367–368.

[51] A. Ramachandran and M. Kantarcioglu, "Smartprovenance: A distributed, blockchain based dataprovenance system," in *Proc. 8th ACM Conf. Data Appl. Secur. Privacy*, 2018, pp. 35–42.

[52] M. Markakis et al., "Sawmill: From logs to causal diagnosis of large systems," in *Proc. Int. Conf. Manage. Data*, 2024, pp. 444–447.

[53] T. Esteves, F. Neves, R. Oliveira, and J. Paulo, "Cat: Content-aware tracing and analysis for distributed systems," in *Proc. 22nd ACM Int. Middleware Conf.*, 2021, pp. 223–235.

[54] Y. Han et al., "Holistic root cause analysis for failures in cloud-native systems through observability data," *IEEE Trans. Serv. Comput.*, vol. 17, no. 6, pp. 3789–3802, Nov./Dec. 2024.

[55] P. Chen, Y. Qi, and D. Hou, "Causeinfer: Automated end-to-end performance diagnosis with hierarchical causality graph in cloud environment," *IEEE Trans. Serv. Comput.*, vol. 12, no. 2, pp. 214–230, Mar./Apr. 2019.

[56] "Datadog - log collection and integrations," Accessed: Feb. 4, 2025. [Online]. Available: https://docs.datadoghq.com/logs/log_collection/?tab=host

[57] "Dynatrace- log analytics," 2023. Accessed: Feb. 4, 2025. [Online]. Available: https://docs.dynatrace.com/docs/analyze-explore-automate/logs

[58] "New relic - get started with log management," 2019. Accessed: Feb. 4, 2025. [Online]. Available: https://docs.newrelic.com/docs/logs/get-started/get-started-log-management/

[59] "Understanding container monitoring: Best practices and tools," 2021. Accessed: Feb. 4, 2025. [Online]. Available: https://www.aquasec.com/cloud-native-academy/docker-container/container-monitoring/#section-2

[60] X. Zhao et al., "lprof: A non-intrusive request flow profiler for distributed systems," in *Proc. 11th USENIX Symp. Operating Syst. Des. Implementation*, 2014, pp. 629–644.

[61] X. Zhao, K. Rodrigues, Y. Luo, D. Yuan, and M. Stumm, "Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle," in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 603–618.

[62] M. Bonola et al., "Faster software packet processing on FPGA NICs with eBPF program warping," in *Proc. USENIX Annu. Tech. Conf.*, 2022, pp. 987–1004.
[63] S. Miano, F. Risso, M. V. Bernal, M. Bertrone, and Y. Lu, "A framework for eBPF-based network functions in an era of microservices," *IEEE Trans. Netw. Service Manag.*, vol. 18, no. 1, pp. 133–151, Mar. 2021.
[64] Y. Zhou, Z. Wang, S. Dharanipragada, and M. Yu, "Electrode: Accelerating distributed protocols with eBPF," in *Proc. 20th USENIX Symp. Networked Syst. Des. Implementation*, 2023, pp. 1391–1407.
[65] Y. Zhong et al., "XRP: In-kernel storage functions with eBPF," in *Proc. 16th USENIX Symp. Operating Syst. Des. Implementation*, 2022, pp. 375–393.
[66] M. Jadin, Q. D. Coninck, L. Navarre, M. Schapira, and O. Bonaventure, "Leveraging eBPF to make TCP path-aware," *IEEE Trans. Netw. Service Manage.*, vol. 19, no. 3, pp. 2827–2838, Sep. 2022.
[67] C. Cassagnes, L. Trestioreanu, C. Joly, and R. State, "The rise of eBPF for non-intrusive performance monitoring," in *Proc. IEEE/IFIP Netw. Operations Manage. Symp.*, 2020, pp. 1–7.
[68] Tetragon, "Tetragon| github," Apr. 2023, Accessed: Feb. 5, 2025. [Online]. Available: https://github.com/cilium/tetragon
[69] A. Ahmad, S. Lee, and M. Peinado, "HARDLOG: Practical tamper-proof system auditing using a novel audit device," in *Proc. 43th IEEE Symp. Secur. Privacy*, 2022, pp. 1791–1807.
[70] C. Yao, D. Agrawal, G. Chen, B. C. Ooi, and S. Wu, "Adaptive logging: Optimizing logging and recovery costs in distributed in-memory databases," in *Proc. 17th Int. Conf. Manage. Data*, 2016, pp. 1119–1134.
[71] S. Ma et al., "Kernel-supported cost-effective audit logging for causality tracking," in *Proc. USENIX Annu. Tech. Conf.*, 2018, pp. 241–254.
[72] J. Lockerman et al., "The FuzzyLog: A partially ordered shared log," in *Proc. 13th USENIX Symp. Operating Syst. Des. Implementation*, 2018, pp. 357–372.
[73] S. Boucher, A. Kalia, D. G. Andersen, and M. Kaminsky, "Putting the 'micro' back in microservice," in *Proc. Conf. Usenix Annu. Tech. Conf.*, 2018, pp. 645–650.
[74] S. Wang, Z. Ding, and C. Jiang, "Elastic scheduling for microservice applications in clouds," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 1, pp. 98–115, Jan. 2021.
[75] A. Arora, S. Kulkarni, and M. Demirbas, "Resettable vector clocks," in *Proc. 19th Annu. ACM Symp. Princ. Distrib. Comput.*, 2000, pp. 269–278.
[76] M. Raynal, "About logical clocks for distributed systems," *ACM SIGOPS Operating Syst. Rev.*, vol. 26, no. 1, pp. 41–48, 1992.
[77] V. Estivill-Castro and D. Wood, "A survey of adaptive sorting algorithms," *ACM Comput. Surv.*, vol. 24, no. 4, pp. 441–476, 1992.
[78] Y. Tokusashi, H. T. Dang, F. Pedone, R. Soulé, and N. Zilberman, "The case for in-network computing on demand," in *Proc. 14th ACM EuroSys Conf.*, 2019, pp. 1–16.
[79] N. Hu, Z. Tian, X. Du, and M. Guizani, "An energy-efficient in-network computing paradigm for 6G," *IEEE Trans. Green Commun. Netw.*, vol. 5, no. 4, pp. 1722–1733, Dec. 2021.

Utkalika Satapathy (Member, IEEE) received the master's degree from the International Institute of Information Technology, Bhubaneswar, India. She is currently working toward the PhD degree with the Indian Institute of Technology, Kharagpur, India. Her research interests include distributed systems, operating systems, and observability. Utkalika is a member of the IEEE and IEEE Communications Society.

Harsh Borse received the BTech degree in computer science and engineering from the Acropolis Institute of Technology, India. He is currently working toward the master's degree with the Indian Institute of Technology, Kharagpur, India. His research interests include anomaly detection, distributed systems, and machine learning. Harsh is a member of IEEE and IEEE Communications Society.

Rajat Bachhawat received the BTech degree in computer science and engineering from the Indian Institute of Technology Kharagpur, India, in 2023. He is currently a systems engineer with Quadeye. His research interests include computer networks, operating systems, distributed systems, and tracing technologies.

Neha Dalmia received the bachelor's and master's degrees in computer science and engineering from the Indian Institute of Technology Kharagpur, India. She is currently a systems engineer with Quadeye. Her research interests include distributed systems, low-latency C++ dev, and traveling.

Subhrendu Chattopadhyay received the MTech and PhD degrees from the Indian Institute of Technology Guwahati. He is currently an assistant professor with the Thapar Institute of Engineering and Technology, India. His research interests include computer networks, operating systems, and distributed computing. Dr. Subhrendu is a Member IEEE Communications Society and Association for Computing Machinery

Sandip Chakraborty (Senior Member, IEEE) received the bachelor's degree from Jadavpur University, Kolkata, in 2009, and the MTech and PhD degrees from IIT Guwahati, in 2011 and 2014, respectively. He is currently an associate professor with the Department of Computer Science and Engineering, Indian Institute of Technology Kharagpur and leading the Ubiquitous Networked Systems Lab (UbiNet – https://ubinet-iitkgp.github.io/ubinet/). His research interests include distributed systems, pervasive, ubiquitous, edge computing, and human-computer interactions. He is the Area Editor of *IEEE Transactions on Services Computing*, Elsevier *Ad Hoc Networks*, and Elsevier *Pervasive and Mobile Computing* journals.