XPLOG: A Dynamic Observability Framework for Distributed Sandboxed Microservices

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

***Abstract*—Runtime application observability is crucial not only for system provenance but also for the dynamic orchestration of deployed microservices for 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 the log richness with minimum system overhead while preserving the causality of the 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 in scalable, dynamic, and distributed Information Technology (IT) infrastructure. The core components of these architectures are containers, microservices, and immutable infrastructure, which help build a robust, maintainable, and loosely coupled ecosystem. When integrated with automation, these technologies can help developers perform frequent and predictable changes with little effort. The *Cloud Native Computing Foundation*[1] (CNCF) aims to make cloud-native computing ubiquitous, ensuring that cloud-native computing becomes widely adopted and accessible across several industries and organizations. The panorama of application development, deployment, and management has significantly improved when the microservice architecture is integrated with the DevOps philosophy. Unlike traditional monolithic applications, which incorporate all the functionalities into a unified system, the microservice architecture segregates applications into independent, manageable, and easily deployable components that handle a specific function. That transition has led many organizations to migrate from a monolithic to microservice architecture.

Comprehensive monitoring and observability are essential to ensure better operations as IT infrastructure becomes increasingly distributed. While monitoring focuses on tracking predefined metrics, collecting logs, and identifying symptoms of operational issues, observability [1], [2] provides the contextual understanding needed to investigate those issues and find the root cause by analyzing system behavior and interactions. However, the traditional monitoring and observability tools are inadequate to accommodate the dynamic nature of the distributed microservice architecture and the rapid deployment cycles they need. On top of that, the increasing complexity of modern infrastructure encompassing edge computing, cloud-native deployments, and Industrial Internet of Things (IIoT) configurations, which try to manage diverse sets of elements like users, VMs, and applications connected via a heterogeneous network, necessitates sophisticated solutions for monitoring. As a result, implementing an observability framework for cloud native distributed architecture is vitally important to reduce metrics such as *Mean Time To Repair* or *Mean Time To Recovery* (MTTR) that track the average duration of outages and provide more effective root cause analysis.

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 [3], [4]. 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 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 [5], [6] dynamically, which can further be used for system vulnerability detection [7], [8], root-cause analysis [9]–[11], 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 frame-

Utkalika Satapathy, Harsh Borse, and Sandip Chakraborty are with the Department of Computer Science and Engineering, Indian Institute of Technology, Kharagpur, India (utkalika.satapathy01@gmail.com, harshzf2@gmail.com, sandipc@cse.iitkgp.ac.in). Neha Dalmia and Rajat Bachhawat are with Quadeye (was associated with Indian Institute of Technology Kharagpur during this work, nehadalmia002@gmail.com, rbachhawat.96@gmail.com). Subhrendu Chattopadhyay is with Thapar Institute of Engineering and Technology (TIET), Patiala, India (subhrendu.subho@gmail.com).

This work has been supported by Indira Gandhi Center for Atomic Research (IGCAR).

[1]https://www.cncf.io/ (Accessed: October 10, 2025)

work over a distributed sandboxing environment (i.e., mostly lightweight containers) has the following challenges.

• Challenge ①: *Relevant Log Extraction for Multi-Host Sandboxed Applications.* The sandboxing platforms (like Docker, Kubernetes, etc.) abstract out the underlying execution of applications, making it challenging to access granular application-level behavior. Existing solutions tend to perform application binary analysis (static or dynamic) to extract relevant logging information. However, analyzing binaries is impractical in distributed environments due to two key factors: (i) application binaries are often proprietary, owned by different organizations, and inaccessible, and (ii) binary analysis tools are architecture-dependent, making them unsuitable for heterogeneous environments. Current logging tools lack the ability to provide comprehensive visibility into application-level behavior. Therefore, we need to design a grey-box approach (extracting the relevant information without snooping inside the application) to generate meaningful logs from the running microservices, which becomes challenging as we need to rely solely on the application-generated logs without having any visibility to the actual application source.

• 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], [12], [13] 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 [14], [15]. (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, 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 [16]–[19] 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 eBPF ring 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` [24] social media application with 30 microservices, each with 3-5 replicas, spawning over 10-30 different hosts, and compare its performance with `Tracee` [25], 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 log-

TABLE I: Recent research work for logging framework

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	Francisco et al. [20]	2021	✗	✗	✓	✗	✓	✗	✓	✗	✓
2	Liu et al. [21]	2020	✓	✓	✓	✓	✗	✗	✗	✗	✓
3	Joshua et al. [22]	2020	✓	✗	✓	✓	✗	✗	✗	✗	✓
4	Francisco et al. [23]	2018	✗	✗	✓	✗	✓	✗	✓	✗	✗
5	Our Approach (*XPLOG*)	2025	✓	✓	✓	✓	✓	✓	✓	✓	✓

ging, log analysis, and causality analysis.

A. Logging in Distributed Cloud Infrastructure

Existing approaches for system logging primarily focused on collecting effective logs for system provenance [26]–[29]. These frameworks primarily develop custom kernel modules, either static [1], [30], [31], proxy-based [2], [32] or Loadable Kernel Module (LKM)-based [33], to combine the logs from the applications and the kernel to develop the system provenance graph. Static hooking of modules inside the kernel requires re-compilation of the entire kernel source, and hence, not a robust solution. Proxy or LKM-based approaches can attach the logging modules only at specific hook points; therefore, such solutions are not scalable for large-scale microservice architectures. Also, LKM-based solutions are not robust against kernel version upgrades [34]. *SLEUTH* [35] reconstructs attack scenarios in real-time using a memory-based dependency graph of audit logs, enabling efficient detection, impact analysis, and visualization across multiple platforms. [36] paper proposes a methodology that combines log parsing and machine learning to detect runtime problems in console logs. *ELT* [37] uses both fine-grained and coarse-grained log features for anomaly detection.

B. Provenance Tracking in Cloud

Provenance tracking for the monolithic [1], [30], [32], [38] and microservice [2], [31], [33]-based applications have widely been studied in the literature. In [39], the authors have proposed $PROV_{2R}$, which captures provenance from unstructured processes using records and replay and taint tracking. However, this requires the instrumentation of programs. [40] proposes *Scippa*, an extension to the Android IPC mechanism that provides provenance information to detect attacks. *Hi-Fi* [41] is a kernel-level provenance system leveraging the *Linux Security Modules*(LSM) framework to collect whole-system provenance for forensic analysis. [42] extends document provenance in cloud environments by tracking both editing and exposure events, enabling the identification of potential external information sources. In [43], the authors have proposed a blockchain and smart contracts-based system called *SmartProvenance* to provide privacy-preserving, access-controlled data provenance with proof of change, trust enforcement, and efficient operation. Certain methods are hindered by the dependency explosion problem, where an excessive number of dependencies are generated, complicating analysis and increasing overhead. Others require instrumentation, which involves modifying the application code or environment to collect detailed data. This process is both resource-intensive and impractical for deployment in enterprise settings due to its potential to disrupt normal operations and introduce additional maintenance challenges.

C. Causality and Systems

Past works have also leveraged causality [20], [23], [44]–[46]. However, the results are based on a temporal relationship rather than the causality of the events. In [23], the authors have analyzed the context of network requests and lack the capability for context and content-aware analysis of storage I/O. The major limitation of *CauseInfer* [47] is that it primarily focuses on numerical data and does not support non-numerical data like log events and configuration data.

D. 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* [48],*Dynatrace* [49],*New Relics* [50],*Aqua* [51] 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* [52] and *Stitch* [53] 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.

E. Use of eBPF in Logging

On the other hand, eBPF has been widely studied to extend the kernel functionality from several perspectives, like for fast packet processing [54], network function management [55], accelerating distributed protocols [56], in-kernel storage management [57], path-aware TCP [58], container network observability [21], [59], etc. Given its wide-scale adaptability for various applications on system observability, eBPF can 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 [13], Logstash [12], `Tracee` [25], *Tetragon* [60] etc., are not

suitable for multi-layer logging. Whereas several works in the literature have focused on efficient system auditing [32], [61], [62], they have primarily focused on developing a logging mechanism for a single-host monolithic application. There have been several works in the literature [63]–[65], 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 logs from multiple microservices, which are causally-consistent. 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.
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 [66], [67], 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.

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

XPLOG utilizes eBPF to 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, eBPF enables 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* [68]) 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 eBPF data structures to address this.

As shown in Figure 1, XPLOG comprises (1) the *XPLOG*

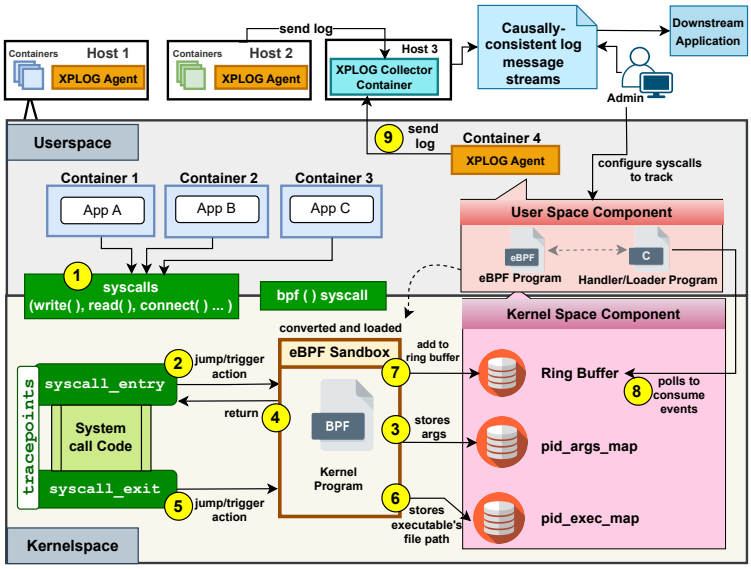


Fig. 1: XPLOG: User and Kernel-space Components

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 containers. 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 eBPF probes 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 eBPF ring 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 eBPF probe 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 & eBPF Probes: We configure Kernel-space tracepoints to hook syscall entry and exit events by executing eBPF probes. 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) eBPF Maps: The maps are used to share data between the user and the kernel-space. We use following eBPF maps: (i) *pid_args_map:* An eBPF map 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 eBPF Buffer to be polled from the user-space component. Consequently, only one log will be pushed to the eBPF buffer 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 eBPF map 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 eBPF ring 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 eBPF ring 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 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 eBPF ring 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

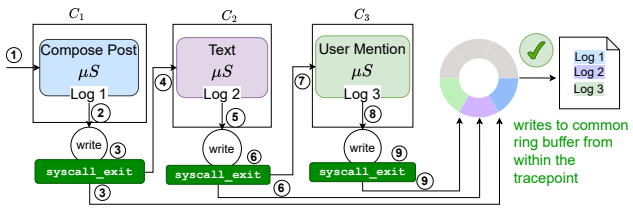


Fig. 2: Using eBPF ring buffer to ensure causal ordering while generating a common log file

eBPF probes 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

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*

file path of the event to the `pid_exec_map` (Fig. 1 ⑥). Consequently, the eBPF ring 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 eBPF maps, 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. 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*" [68] implemented in the kernel space triggered with each cross-host communication event. 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 eBPF map. 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 [68]. 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` [25], a widely used system auditing framework that uses eBPF to 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`.

A. Implementation Details

The source code, documentation, and the experimental configuration scripts of `XPLOG` implementation have been open-sourced[2]. During implementation, we have used `C` with `libbpf` library to realize eBPF probe programs. We monitor 19 syscalls across file I/O, process, and socket types (Table III), which are configurable at runtime. We have developed corresponding eBPF probes for each of them separately. Each syscall-generated log contains the fields listed in Table II. These fields are obtained using eBPF helper 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 eBPF instrumentation 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 eBPF ring buffer (described in Section IV-B3) size as 1MB with a polling interval of 50ms, which signifies that after each 50ms, the logs stored in the ring buffer will be forwarded to the `XPLOG` collector. Additionally, we have reserved 2MB for `pid_exec_map` and 64KB for `pid_args_map`.

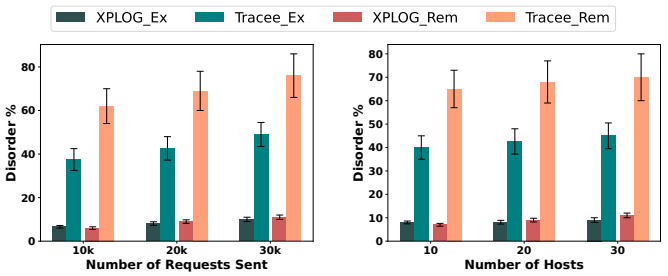
B. Experimental Setup

To evaluate the efficacy of `XPLOG`, we use DeathStarBench [24], an open-source benchmark suite for cloud microservices. We have used the *Social Network* microservice of DeathStarBench as it is the only end-to-end stable application available 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 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

[2]https://anonymous.4open.science/r/XPLOG-PluggableLogging-7457/

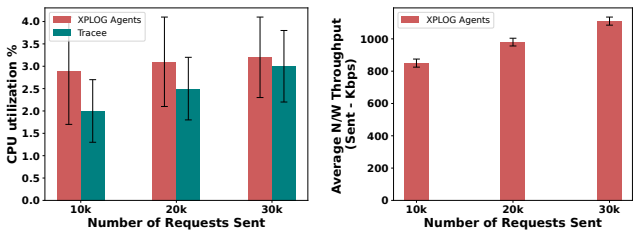
[3]https://github.com/delimitrou/DeathStarBench/raw/master/socialNetwork/figures/socialNet_arch.png (All URLs in this paper have accessed last on October 10, 2025)

[4]https://github.com/wg/wrk



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

Fig. 3: Log disorder between `Tracee` and `XPLOG` (Service endpoint: CP+UT+HT)



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

Fig. 4: Average CPU and network utilization of 19 `XPLOG` agents

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 $64GB$ 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.

For comparison, we collect logs using (a) `Tracee` [25] and (b) the proposed `XPLOG`. Notably, `Tracee` also uses eBPF to 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 have collected `XPLOG`-generated and `Tracee`-generated logs while sending variable numbers of $(10,000\text{-}30,000)$ parallel and asynchronous requests to the target application spanned over multiple hosts without having any explicit clock synchronization. We have also collected the logs by randomly varying the requests and service endpoints to replicate a real-world user interaction.

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 [69]: (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. 3(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. Notably, for $30,000$ parallel requests for the same set of microservices, `Tracee` generated almost $3.5GB$ and comparatively `XPLOG` generates $5.7GB$ of log messages in total (more details in Section V-E, Table V). Similarly, Fig. 3(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 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

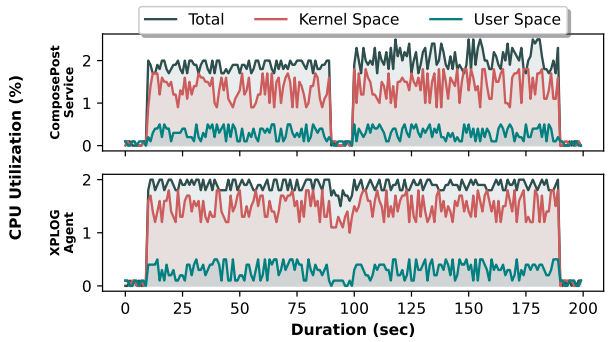
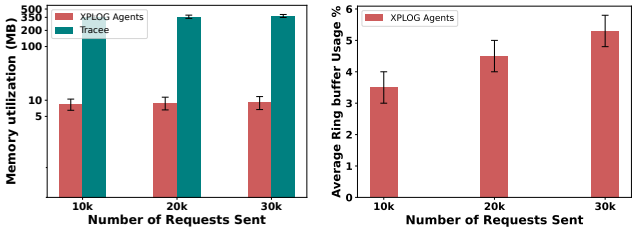


Fig. 5: CPU utilization per request (Service endpoint: CP)



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

Fig. 6: Average Memory overhead and eBPF buffer usage of 19 XPLOG Agents

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.

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

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: Fig. 4 shows the average CPU and network utilization of 19 XPLOG agents running in the workers. We observe that the average CPU usage (see Fig. 4a) of XPLOG Agents are 2.9%, 3.10%, and 3.15% when $10K$, $20K$, and $30K$ requests are sent concurrently. In Fig. 5, 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 eBPF programs; 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. 4b) of the XPLOG agents, which is $\approx 1.7Mbps$, because of the constant streaming of the generated log messages to the collector.

2) Memory Overhead: We observe that the memory overhead (see Fig. 6a) of XPLOG Agents remains constant with $10K$, $20K$, and $30K$ 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 $5.7GB$ when the number of parallel requests is $10,000$ whereas Tracee generates $2.1GB$ of log messages. Notably, this total size of the log messages increased to $5.2GB$ and $13GB$ 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.

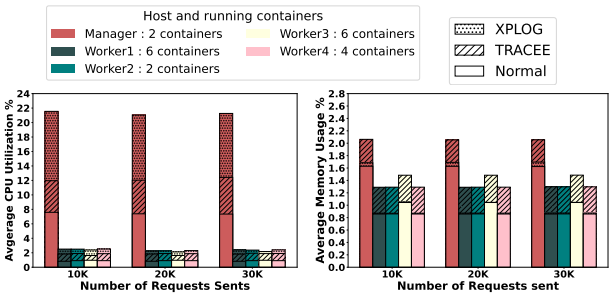
TABLE V: Log sizes vs #Requests (Service endpoint: CP)

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

XPLOG can handle this high amount of logs reasonably well, which can be justified by Fig. 6a 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., ≈ 2/3rd of all logs) of the generated logs are due to the sandboxing environment and platform. Rather 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 eBPF ring buffer while varying the number of requests from $10K$ to $30K$ is presented in Fig. 6b. We have kept the eBPF ring buffer limit as $1MB$, which allows storage for up to $256 * 1024$ log entries. The

[5]https://github.com/google/cadvisor
[6]https://github.com/prometheus/node_exporter



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

Fig. 7: Host Resource utilization (Service endpoint: CP+UT+HT)

buffer is polled every $50ms$. The total experimental duration for $30K$ requests requires ≈ 10 minutes. We observe that the peak utilization of eBPF ring buffer is 5.2% for CP, UT, and HT endpoints.

3) 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. 7). 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. 7a). However, with `XPLOG`, it goes up to 21% (shown in dotted of Fig. 7) as the collector continuously consumes the log messages sent by the agents. However, we observe that the memory usage (shown in Fig. 7b) 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 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 [70], [71]. 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.

TABLE VI: Log writing latency (#Concurrent requests: $10,000$)

Type of Requests	Log entries ($\approx$) 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

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 0.5ms. 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 1s, indicating that the proposed observability framework makes the log messages available to the downstream applications reasonably fast.

```
/* Application Log */
Host 1 : {
    "event_context": {
        "ts": XXX,"datetime": "XXX",
        "task_context": {
            "host_pid": 314463, "host_tid": 317863,
            "task_command" : "ComposePostServ" ··· }},
    "data": {
        "lms": "[2024-Jul-26 15:33:55.832728] <info>: (
     ComposePostHandler.h:370: ComposePost) Request
     Order : 1, ID : 899493982365583360"},
    "artifacts": {"exe": "/usr/local/bin/
     ComposePostService"}}
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" } },
/* System Log */
Host 1: {
    "event_context": {
        "ts": XXX, "datetime": "XXX",
        "syscall_id": 42,"syscall_name": "connect",
        "task_context": {
        "host_pid": 314463, "host_tid": 317863, ···}},
    "arguments": { "uservaddr":"0x80003760","addrlen
     ":16},
    "artifacts": {"exe": "/usr/local/bin/
     ComposePostService","IP":"10.11.0.63","port
     ":"33315"}}
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" }}}, ···
Host 1 : { ··· "syscall_name" : "send",  "task_context"
     : { "host_pid" :  314463, "host_tid" :  317863,
     ··· }}
Host 2 : { ··· "syscall_name" : "recv",  "task_context"
     : { "host_pid" :  1532108, "host_tid" :  1535513,
     ··· }},
```

Listing 1: Example of `XPLOG`-generated log entries

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

[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.

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	✗	✓

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

[8]https://github.com/usatpath01/Pluggable_Logging/tree/main/scripts

```
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]
4 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>
5 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]
6 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]
7 ...
```

Listing 2: Tracee-generated syscall log snippet

VI. Conclusion

This paper introduced an innovative and non-invasive platform observability framework called XPLOG that leverages eBPF to generate comprehensive logs for distributed edge applications. 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. However, there are several ways through which the robustness and efficacy of XPLOG can be improved further. For example, XPLOG does not work with hosts running multiple different operating systems and, thus, is not suitable for a heterogeneous platform in its present form. Nevertheless, we believe that XPLOG can work with a large set of distributed edge computing applications while opening a new scope for further research in other directions.

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 *27th NDSS*, 2020.

[2] P. Datta, I. Polinsky, M. A. Inam, A. Bates, and W. Enck, "ALASTOR: Reconstructing the provenance of serverless intrusions," in *31st USENIX Security*, 2022.

[3] W. Kellerer, P. Kalmbach, A. Blenk, A. Basta, M. Reisslein, and S. Schmid, "Adaptable and data-driven softwarized networks: Review, opportunities, and challenges," *Proceedings of the IEEE*, vol. 107, no. 4, pp. 711–731, 2019.

[4] M. Scrocca, R. Tommasini, A. Margara, E. D. Valle, and S. Sakr, "The kaiju project: enabling event-driven observability," in *14th ACM DEBS*, 2020.

[5] S. Zawoad, R. Hasan, and K. Islam, "Secprov: Trustworthy and efficient provenance management in the cloud," in *IEEE INFOCOM*, 2018.

[6] T. Blount, A. Chapman, M. Johnson, and B. Ludascher, "Observed vs. possible provenance (research track)," in *13th USENIX TaPP*, 2021.

[7] M. Zipperle, F. Gottwalt, E. Chang, and T. Dillon, "Provenance-based intrusion detection systems: A survey," *ACM Computing Surveys*, vol. 55, no. 7, pp. 1–36, 2022.

[8] M. M. Anjum, S. Iqbal, and B. Hamelin, "ANUBIS: a provenance graph-based framework for advanced persistent threat detection," in *37th ACM/SIGAPP SAC*, 2022.

[9] A. Chen, Y. Wu, A. Haeberlen, W. Zhou, and B. T. Loo, "Differential provenance: Better network diagnostics with reference events," in *14th ACM HOTNETS*, 2015.

[10] 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 *29th NDSS*, 2022.

[11] 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 *28th ACM ASPLOS*, 2023.

[12] Elastic, "Logstash: Collect, parse, transform logs | elastic," https://www.elastic.co/logstash/, June 2024, [accessed 4. Feb. 2025].

[13] Fluent, "Fluentd | open source data collector | unified logging layer," https://www.fluentd.org/, June 2024, [accessed 4. Feb. 2025].

[14] X. Merino and C. E. Otero, "The cost of virtualizing time in linux containers," in *8th IEEE Cloud Summit*, 2022.

[15] M. Cinque, R. Della Corte, and A. Pecchia, "Microservices monitoring with event logs and black box execution tracing," *IEEE Transactions on Services Computing*, vol. 15, no. 1, pp. 294–307, 2019.

[16] M. A. Vieira *et al.*, "Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications," *ACM Computing Surveys*, vol. 53, no. 1, pp. 1–36, 2020.

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

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

[19] T. L. Foundation, "The state of ebpf," https://www.linuxfoundation.org/hubfs/eBPF/The_State_of_eBPF.pdf, January 2024, [accessed 4. Feb. 2025].

[20] F. Neves, N. Machado, R. Vilaça, and J. Pereira, "Horus: Non-intrusive causal analysis of distributed systems logs," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2021.

[21] C. Liu, Z. Cai, B. Wang, Z. Tang, and J. Liu, "A protocol-independent container network observability analysis system based on eBPF," in *IEEE 26th ICPADS*, 2020.

[22] J. Levin and T. A. Benson, "Viperprobe: Rethinking microservice observability with ebpf," in *9th IEEE CloudNet*, 2020.

[23] F. Neves, N. Machado *et al.*, "Falcon: A practical log-based analysis tool for distributed systems," in *48th IEEE/IFIP DSN*, 2018.

[24] Y. Gan *et al.*, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *24th ACM ASPLOS*, 2019.

[25] A. Security, "Tracee| github," https://github.com/aquasecurity/tracee, June 2024, [accessed 4. Feb. 2025].

[26] M. A. Inam *et al.*, "Sok: History is a vast early warning system: Auditing the provenance of system intrusions," in *44th IEEE S&P*, 2023.

[27] D. Huye, Y. Shkuro, and R. R. Sambasivan, "Lifting the veil on Meta's microservice architecture: Analyses of topology and request workflows," in *USENIX ATC*, 2023.

[28] B. Debnath *et al.*, "LogLens: A real-time log analysis system," in *38th IEEE ICDCS*, 2018.

[29] K. Suo, Y. Zhao, W. Chen, and J. Rao, "vnettracer: Efficient and programmable packet tracing in virtualized networks," in *38th IEEE ICDCS*, 2018.

[30] X. Ge, B. Niu, and W. Cui, "Reverse debugging of kernel failures in deployed systems," in *USENIX ATC*, 2020.

[31] S. Y. Lim, B. Stelea, X. Han, and T. Pasquier, "Secure namespaced kernel audit for containers," in *12th ACM SoCC*, 2021.

[32] J. Zeng *et al.*, "WATSON: Abstracting behaviors from audit logs via aggregation of contextual semantics," in *28th NDSS*, 2021.

[33] U. Satpathy, R. Thakur, S. Chattopadhyay, and S. Chakraborty, "DisPro-Track: Distributed provenance tracking over serverless applications," in *IEEE INFOCOM*, 2023.

[34] L. Zhou, F. Zhang, K. Leach, X. Ding, Z. Ning, G. Wang, and J. Xiao, "Hardware-assisted live kernel function updating on intel platforms," *IEEE Transactions on Dependable and Secure Computing*, vol. 21, no. 4, pp. 2085–2098, 2024.

[35] M. N. Hossain, S. M. Milajerdi, J. Wang, B. Eshete, R. Gjomemo, R. Sekar, S. Stoller, and V. Venkatakrishnan, "Sleuth: Real-time attack scenario reconstruction from cots audit data," in *26th USENIX Security*, 2017.

[36] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *22nd ACM SOSP*, 2009.

[37] K. Kc and X. Gu, "Elt: Efficient log-based troubleshooting system for cloud computing infrastructures," in *30th IEEE SRDS*, 2011.

[38] F. Yang, J. Xu, C. Xiong, Z. Li, and K. Zhang, "PROGRAPHER: An anomaly detection system based on provenance graph embedding," in *32nd USENIX Security*, 2023.

[39] M. Stamatogiannakis, E. Athanasopoulos, H. Bos, and P. Groth, "Prov 2r: practical provenance analysis of unstructured processes," *ACM Transactions on Internet Technology (TOIT)*, vol. 17, no. 4, pp. 1–24, 2017.

[40] M. Backes, S. Bugiel, and S. Gerling, "Scippa: System-centric ipc provenance on android," in *30th ACM ACSAC*, 2014.

[41] D. J. Pohly, S. McLaughlin, P. McDaniel, and K. Butler, "Hi-fi: collecting high-fidelity whole-system provenance," in *28th ACM ACSAC*, 2012.

[42] C. Collberg, A. Gibson, S. Martin, N. Shinde, A. Herzberg, and H. Shulman, "Provenance of exposure: Identifying sources of leaked documents," in *IEEE CNS*, 2013.

[43] A. Ramachandran and M. Kantarcioglu, "Smartprovenance: A distributed, blockchain based dataprovenance system," in *8th ACM CODASPY*, 2018.

[44] M. Markakis, A. B. Chen, B. Youngmann, T. Gao, Z. Zhang, R. Shahout, P. B. Chen, C. Liu, I. Sabek, and M. Cafarella, "Sawmill: From logs to causal diagnosis of large systems," in *ACM SIGMOD/PODS*, 2024.

[45] T. Esteves, F. Neves, R. Oliveira, and J. Paulo, "Cat: Content-aware tracing and analysis for distributed systems," in *22nd ACM International Middleware Conference*, 2021.

[46] Y. Han, Q. Du, Y. Huang, P. Li, X. Shi, J. Wu, P. Fang, F. Tian, and C. He, "Holistic root cause analysis for failures in cloud-native systems through observability data," *IEEE Transactions on Services Computing*, 2024.

[47] P. Chen, Y. Qi, and D. Hou, "Causeinfer: Automated end-to-end performance diagnosis with hierarchical causality graph in cloud environment," *IEEE Transactions on services computing*, vol. 12, no. 2, pp. 214–230, 2016.

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

[49] "Dynatrace- log analytics," https://docs.dynatrace.com/docs/analyze-explore-automate/logs, [accessed 4. Feb. 2025].

[50] "New relic - get started with log management," https://docs.newrelic.com/docs/logs/get-started/get-started-log-management/, [accessed 4. Feb. 2025].

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

[52] X. Zhao, Y. Zhang, D. Lion, M. F. Ullah, Y. Luo, D. Yuan, and M. Stumm, "lprof: A non-intrusive request flow profiler for distributed systems," in *11th USENIX OSDI)*, 2014.

[53] 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 *12th USENIX OSDI*, 2016.

[54] M. Bonola *et al.*, "Faster software packet processing on FPGA NICs with eBPF program warping," in *USENIX ATC*, 2022.

[55] S. Miano *et al.*, "A framework for eBPF-based network functions in an era of microservices," *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 133–151, 2021.

[56] Y. Zhou, Z. Wang, S. Dharanipragada, and M. Yu, "Electrode: Accelerating distributed protocols with eBPF," in *20th USENIX NSDI*, 2023.

[57] Y. Zhong *et al.*, "XRP: In-kernel storage functions with eBPF," in *16th USENIX OSDI*, 2022.

[58] M. Jadin, Q. De Coninck, L. Navarre, M. Schapira, and O. Bonaventure, "Leveraging eBPF to make TCP path-aware," *IEEE Transactions on Network and Service Management*, vol. 19, no. 3, pp. 2827–2838, 2022.

[59] C. Cassagnes, L. Trestioreanu, C. Joly, and R. State, "The rise of ebpf for non-intrusive performance monitoring," in *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*, 2020.

[60] Tetragon, "Tetragon | github," https://github.com/cilium/tetragon, April 2023, [accessed 5. Feb. 2025].

[61] A. Ahmad, S. Lee, and M. Peinado, "Hardlog: Practical tamper-proof system auditing using a novel audit device," in *43th IEEE S&P*, 2022.

[62] R. Sekar, H. Kimm, and R. Aich, “eaudit: A fast, scalable and deployable audit data collection system,” in *45th IEEE S&P*, 2024.
[63] 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 *17th IEEE MDM*, 2016.
[64] S. Ma *et al.*, “Kernel-Supported Cost-Effective audit logging for causality tracking,” in *USENIX ATC*, 2018.
[65] J. Lockerman *et al.*, “The FuzzyLog: A partially ordered shared log,” in *13th USENIX OSDI*, 2018.
[66] S. Boucher, A. Kalia, D. G. Andersen, and M. Kaminsky, “Putting the "micro" back in microservice,” in *USENIX ATC*, 2018.
[67] S. Wang, Z. Ding, and C. Jiang, “Elastic scheduling for microservice applications in clouds,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 1, pp. 98–115, 2020.
[68] A. Arora, S. Kulkarni, and M. Demirbas, “Resettable vector clocks,” in *19th ACM PODC*, 2000.
[69] V. Estivill-Castro and D. Wood, “A survey of adaptive sorting algorithms,” *ACM CSUR*, vol. 24, no. 4, pp. 441–476, 1992.
[70] Y. Tokusashi, H. T. Dang, F. Pedone, R. Soulé, and N. Zilberman, “The case for in-network computing on demand,” in *14th ACM EuroSys*, 2019.
[71] N. Hu, Z. Tian, X. Du, and M. Guizani, “An energy-efficient in-network computing paradigm for 6g,” *IEEE Transactions on Green Communications and Networking*, vol. 5, no. 4, pp. 1722–1733, 2021.

Utkalika Satapathy has completed her Master’s degree from International Institute of Information Technology Bhubaneswar, India. Currently, Utkalika is pursuing her PhD degree at the Indian Institute of Technology, Kharagpur, India. Her research interests encompass Distributed Systems, Operating Systems and Observability. Utkalika is a Member of IEEE, IEEE Communications Society.

Harsh Borse received the B.Tech degree in Computer Science and Engineering from Acropolis Institute of Technology, India. Currently, he is pursuing his Master’s degree from the Indian Institute of Technology, Kharagpur, India. His research interests include Anomaly Detection, Distributed Systems, and Machine Learning. Harsh is a Member of IEEE, IEEE Communications Society.

Rajat Bachhawat received the B.Tech. degree in Computer Science and Engineering from the Indian Institute of Technology Kharagpur, India, in 2023. He is interested in computer networks, operating systems, distributed systems, and tracing technologies.

Neha Dalmia has completed her Bachelors and Masters in Computer Science and Engineering from Indian Institute of Technology Kharagpur, India. She is interested in distributed systems, low latency C++ dev, and traveling.

Subhrendu Chattopadhyay received the M.Tech and PhD degree from the Indian Institute of Technology Guwahati. At present, he is working as an Assistant Professor at Thapar Institute of Engineering and Technology, India. His research interests include Computer networks, Operating Systems, and distributed computing. Dr. Subhrendu is a Member of IEEE, IEEE Communications Society, and Association for Computing Machinery

Sandip Chakraborty (Senior Member, IEEE) is working as an Associate Professor in the Department of Computer Science and Engineering at the Indian Institute of Technology Kharagpur, and leading the Ubiquitous Networked Systems Lab (UbiNet, https://ubinet-iitkgp.github.io/ubinet/). He obtained his Bachelor’s degree from Jadavpur University, Kolkata in 2009 and M Tech and Ph.D., both from IIT Guwahati, in 2011 and 2014, respectively. The primary research interests of Dr. Chakraborty are Distributed Systems, Pervasive, Ubiquitous and Edge Computing, and Human-Computer Interactions. He received various awards including the Google Academic Research Award (GARA) 2024, Google Award for Inclusion Research on Societal Computing 2023, Indian National Academy of Engineering (INAE) Young Engineers’ Award 2019, and the Honorable Mention Award in IEEE MDM 2022, ACM SIGCHI EICS 2020. Dr. Chakraborty is one of the founding chairs of ACM IMOBILE, the ACM SIGMOBILE Chapter in India. He is working as an Area Editor of IEEE Transactions on Services Computing, Elsevier Ad Hoc Networks and Elsevier Pervasive and Mobile Computing journals.