PROVENANCE AND OBSERVABILITY FRAMEWORKS FOR SECURE DISTRIBUTED MICROSERVICES

Utkalika Satapathy

PROVENANCE AND OBSERVABILITY FRAMEWORKS

FOR SECURE DISTRIBUTED MICROSERVICES

Thesis submitted to the
Indian Institute of Technology, Kharagpur
For award of the degree

of

Doctor of Philosophy

by

Utkalika Satapathy

Under the supervision of

Prof. Sandip Chakraborty

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

April 2026

APPROVAL OF THE VIVA-VOCE BOARD

Date: / / 20

Certified that the thesis entitled **"Provenance and Observability Frameworks for Secure Distributed Microservices"** submitted by Utkalika Satapathy to the Indian Institute of Technology, Kharagpur, for the award of the degree of Doctor of Philosophy has been accepted by the external examiners and that the student has successfully defended the thesis in the viva-voce examination held today.

(Member of DSC) (Member of DSC) (Member of DSC)

(Supervisor) (Joint Supervisor)

(External Examiner) (Chairman)

CERTIFICATE

This is to certify that the thesis entitled ***"Provenance and Observability Frameworks for Secure Distributed Microservices"****, submitted by Utkalika Satapathy to the Indian Institute of Technology, Kharagpur, for the partial fulfillment of the award of the degree of Doctor of Philosophy in Computer Science and Engineering, is a record of bona fide research work carried out by her under my supervision and guidance. The thesis in my opinion, is worthy of consideration for the award of the degree of Doctor of Philosophy in accordance with the regulations of the Institute. To the best of my knowledge, the results embodied in this thesis have not been submitted to any other University or Institute for the award of any other Degree or Diploma.*

Sandip Chakraborty
Associate Professor
Department of Computer
Science and Engineering,
IIT Kharagpur

Date:

Date:

DECLARATION

I certify that

a. The work contained in this thesis is original and has been done by me under the guidance of my supervisors.

b. The work has not been submitted to any other Institute for any degree or diploma.

c. I have followed the guidelines provided by the Institute in preparing the thesis.

d. I have conformed to the norms and guidelines given in the Ethical Code of Conduct of the Institute.

e. Whenever I have used materials (data, theoretical analysis, figures, and text) from other sources, I have given due credit to them by citing them in the text of the thesis and giving their details in the references.

f. Whenever I have quoted written materials from other sources, I have put them under quotation marks and given due credit to the sources by citing them and giving required details in the references.

Utkalika Satapathy

Dedicated to my family, in-laws, friends, husband
&
in loving memory of my mother

Their sacrifices motivated me to complete my journey

ACKNOWLEDGMENTS

The completion of this thesis is more than just an academic milestone; it marks the culmination of a deeply personal journey filled with growth, learning, and transformation. Along the way, I have not only gained knowledge in my chosen field but also collected countless memories, moments of reflection, and life lessons that will remain with me forever. This acknowledgement is an opportunity to pause and look back at the many instances of encouragement, the days of self-doubt, the quiet victories, and the enduring support of those who stood by me. In those very moments, I found the strength to continue. I am profoundly grateful to everyone who has played a part in this journey, especially to those whose presence, kindness, and belief in me have made this achievement possible.

First and foremost, I extend my deepest gratitude to my supervisor, Prof. Sandip Chakraborty, for his constant support, insightful guidance, and persistent encouragement throughout my doctoral journey. My first interaction with him was during my master's internship at IIT Kharagpur under his supervision, mentored by Soumyajit da. I clearly remember reaching out to him again during the COVID period to inquire about an open position in a research project. His prompt and encouraging response paved the way for the beginning of my research journey as a Junior Research Fellow (JRF) under his supervision. He has been much more than an academic advisor; he has been a mentor in every sense. During times of uncertainty, his advice offered clarity; during challenging phases, his faith in me provided strength. I feel incredibly fortunate to have had such a dedicated and compassionate

mentor who has profoundly shaped both my academic path and personal growth.

I would also like to extend my sincere thanks to Prof. Niloy Ganguly, Head of the Department of Computer Science and Engineering, for his support and for cultivating an intellectually stimulating environment that inspires innovation and excellence. I am equally grateful to Prof. Arobindo Gupta, former Head of the Department and a member of my Doctoral Scrutiny Committee (DSC), for his invaluable feedback and constant encouragement. I owe special thanks to all the members of my DSC: Prof. Shamik Sural (Chairman), Prof. Soumya Kanti Ghosh, and Dr. Deborati Sen (from the G.S. Sanyal School of Telecommunication), for their insightful suggestions and constructive criticism, which have significantly contributed to the quality and direction of my thesis. Their probing questions and thoughtful insights challenged me to approach my research with greater depth and precision.

I am immensely thankful to the Department of Computer Science and Engineering at IIT Kharagpur for providing both the academic foundation and supportive research environment that have been instrumental to my journey. The department has been my academic home over the past four years, and I deeply appreciate the guidance and encouragement extended by all faculty members and staff. I am especially grateful to Sadhan da, Bappa da, Sahoo da, Durga da, Prasun da, and all other staff members for their timely assistance in navigating various administrative procedures, often with great efficiency and care.

I would also like to thank all the anonymous reviewers of my publications, whose valuable feedback helped improve the quality and impact of this thesis. I am truly grateful to the Indira Gandhi Centre for Atomic Research (IGCAR), my project sponsor, for their generous support toward my doctoral research. I also acknowledge IEEE INFOCOM, ACM IARCS, COMSNETS LRN, and the Government of India for providing financial assistance and the opportunity to present

my work at leading international forums.

I consider myself extremely fortunate to be a part of UbiNet Lab. It's not just a place where I worked; it became like a second home to me. The lab was full of energy, support, and great conversations over tea. Being around such a wonderful group of people helped me grow a lot, both as a researcher and as a person. I am grateful to all my labmates for their constant support and friendship. Sugandh was the first person who introduced me to the lab when I was allotted a desk in Takshashila. At that time, there was no space in the SMR lab, so I used to sit alone in the Project Lab. I still remember how she made me feel welcome from the very first day. She helped me get a seat in the SMR lab so I could be more connected with the team. In many ways, she filled that gap and acted as the bridge between me and the rest of the lab. Her support during those early days meant a lot to me. Argha brought so much life to the lab. He often sang with his headphones on, playing catchy songs that filled the lab with joy and energy. Lalan made the lab lively with his fun personality. Gunjan never failed to make us laugh with his witty sense of humor and playful nature. Prasenjit is always ready to help with any research problem or technical query. I also want to thank Debasree, Salma, Tirthankar Sengupta, Rajkumar ji, and Subhankar for their wonderful companionship. Our endless discussions and tea-break sessions were both enlightening and fun. I am equally thankful to my seniors, Soumyajit da and Bishakh, for their guidance during the early stages of my research journey. It was always fun to work at night when Anirban da would play the guitar, filling the quiet hours with music and making the lab feel alive. I would also like to express my heartfelt appreciation to my juniors Aritra, Neha, Shailesh, Arup, Tirthankar Halder, Debjit, Manjeet, Sayantan, and Rajib for their infectious enthusiasm and contributions. Aritra is like a little brother to me. He loves teasing me, and our friendly banter always brought a smile to my face. I remember Aritra

asking me to get a signed copy of the book my father wrote. It was such a sweet and thoughtful thing to do, and it really meant a lot to me. And, Neha, with her never-ending series of questions, always kept things interesting, and honestly, I've always found her genuinely good and sincere. Working with them was an enriching experience, and I wish them all the success in their future endeavors. One of the most memorable milestones of my Ph.D. journey was my first trip abroad to attend IEEE INFOCOM in New Jersey, United States. I had the privilege of traveling with Prof. Sandip Chakraborty, who made the experience even more special. I fondly remember him saying, "Give me a list of places you'd like to visit, and I'll try my best to cover them." Staying true to his word, we visited nearly all the destinations on my list, including Princeton and Rutgers University. His friendly and approachable nature made the trip unforgettable. Without a doubt, he is one of the coolest and most supportive supervisors one could ask for.

I would like to thank my collaborators, Harsh Borse (CNeRG Lab, IIT Kharagpur) and Dr. Subhrendu Chattopadhyay (Assistant Professor, Thapar Institute of Engineering and Technology, Patiala), for their valuable insights and joint efforts in our research endeavors. I am deeply thankful to Harsh, with whom I have collaborated on most of my research work. I have learned a great deal from him, both academically and personally. His lively spirit, positivity, and unwavering enthusiasm have made every collaboration enjoyable and enriching. I am sincerely thankful to Dr. Subhrendu Chattopadhyay for his invaluable guidance and mentorship. He played a pivotal role in teaching me how to structure and write research papers and has been instrumental in shaping the direction of my work through his insightful advice and constant support.

I have had the opportunity to mentor and collaborate with exceptional student researchers. I would like to thank my MTP students: Ishan Sharma, Vamshidhar

Reddy, Neha Dalmia, Sri Ram Chowdary Ravi, Ashwamegh Rathore, and Parth Tusham; and my BTP students: Rishabh Thakur, Rajat Bachhawat, Nandini Jalan, Narayan Gupta, Nisarg, Divyansh Vijayvergia, Shivam Raj, and Shivansh Shukla. I also extend my thanks to my intern, Shreya Sinha, and Arijit Sarkar for their dedication and valuable contributions to our projects. Apart from the lab members, I am also thankful to Tishya, Debjyoti, Soumya, Niskal, Owais, Sourjyadip, Alka, Sujeet, and Harshita for such nice interactions.

Thank you to Anisha, Anshita, Aishi, Manpreet, Pallavi, and Amrita, my lovely hostelmates from VSRC-2. Your friendship, constant support, and all the memories we created together have been a true source of strength and comfort throughout this journey. Anisha, in particular, holds a special place in this journey. She was my first friend at IIT Kharagpur during the COVID era when classes were online. Our bond only deepened after we returned to campus, as we lived in the same hostel. Her cheerful and lively nature and our endless gossip sessions added so much warmth and happiness to my daily life.

Beyond the academic domain, I am deeply grateful to all my friends, especially my close friends Meenu and Pranali, who were also my M.Tech roommates. Their friendship has been a constant pillar throughout this journey. In every moment of doubt or joy, they were there to listen, encourage, and celebrate with me. Their presence brought much-needed balance, laughter, and emotional support.

This thesis is dedicated to the most important constants in my life: my family, my in-laws, and my husband, Prabhu. I owe everything to my father, Dasarathi Satapathy, who has been a source of strength through every stage of my life. He has stood by my decisions, encouraged me during difficult times, and guided me with the wisdom of both a parent and a teacher. His quiet resilience and unwavering presence gave me the confidence to persevere through every challenge. I am also deeply

thankful to my in-laws for their patience, understanding, and belief in my career and aspirations. Their support allowed me to pursue this demanding journey wholeheartedly. Their patience and faith in me have meant more than words can express.

And to my husband, Prabhu, my best friend, my biggest cheerleader, and my strongest support. Thank you for being by my side through every late night, every setback, and every small win. This thesis would not exist without your love, your steady encouragement, and your endless belief in me. You've always been there to bring calm in chaos, laughter in stress, and perspective when I needed it most. The way you've balanced everything and stood beside me through these years is something I'll carry in my heart forever. I truly couldn't have done this without you.

Lastly, I dedicate this thesis to the cherished memory of my late mother, Ranjita Kar. Though she is no longer with me, her love, values, and quiet strength have continued to guide me every step of the way. Her absence has been deeply felt, but her presence lives on in everything I do, and her blessings have carried me through this journey.

Utkalika Satapathy

IIT Kharagpur, India

Author's Biography

Utkalika Satapathy received her B.Tech. degree in Information Technology from Silicon Institute of Technology, Odisha, India, in 2013. She worked at Infosys Ltd. as a Senior System Engineer from 2013 to 2017 before pursuing her M.Tech. degree at the International Institute of Information Technology, Bhubaneswar, India, which she completed in 2019. From 2019 to 2021, she served as a contractual faculty member at the College of Engineering and Technology (CET) (now Odisha University of Technology and Research (OUTR)). Her research interests lie in Computing and Distributed Systems, particularly focusing on Observability.

Thesis Related Publications

1. **Utkalika Satapathy**, Rishabh Thakur, Subhrendu Chattopadhyay, Sandip Chakraborty, "Disprotrack: Distributed provenance tracking over serverless applications", in proc. of 42^{nd} IEEE Conference on Computer Communications (IEEE INFOCOM), 2023, New York area, USA
2. **Utkalika Satpathy**, Harsh Borse, Sandip Chakraborty, "Towards Generating a Robust, Scalable and Dynamic Provenance Graph for Attack Investigation over Distributed Microservice Architecture",in proc. of 17^{th} International Conference on Communication Systems & Networks (COMSNETS), 2025, Bangalore, India. (Received Best Paper Runner's Up Award)
3. **Utkalika Satapathy**, Harsh Borse, Rajat Bachhawat, Neha Dalmia, Subhrendu Chattopadhyay, and Sandip Chakraborty, "XPLOG: A Dynamic Observability Framework for Distributed Sandboxed Microservices", in IEEE Transactions on Services Computing (IEEE TSC), 2025. [Major Revision]
4. **Utkalika Satapathy**, Harsh Borse and Sandip Chakraborty, "*μProvGAE*: Context-Enriched Provenance Graph for Attack Detection in Distributed Systems using Graph Autoencoders, in IEEE Transactions on Services Computing (IEEE TSC), 2026. [Submitted]

ABSTRACT

The rapid evolution of cloud-native architectures has led to the widespread adoption of microservices and serverless computing, significantly enhancing system scalability, modularity, and deployment agility. However, these distributed, sandboxed, and ephemeral environments introduce severe challenges in system observability, forensic investigation, and security monitoring. Traditional monitoring and audit mechanisms that have been designed primarily for monolithic applications are rendered inadequate in these contexts due to fragmented visibility, namespace isolation, asynchronous communication, and the absence of unified log semantics. As a result, detecting advanced persistent threats (APTs), tracing causality across services, or conducting root cause analysis becomes significantly hindered in practice.

This thesis addresses the core research problem of enabling scalable, non-intrusive, and causally-consistent observability in distributed microservice ecosystems. It proposes a comprehensive approach to bridge the semantic and operational gap between system-level telemetry and application-level behavior by constructing unified provenance graphs that encode fine-grained, temporally and causally ordered interactions across heterogeneous components. These provenance graphs serve as the foundational structure for understanding system dynamics, detecting anomalies, and investigating multi-stage attacks.

The first contribution of the thesis develops methods to statically extract control flow representations from microservice binaries and dynamically associate them with system call logs during runtime, thereby capturing semantically meaningful relationships between logs originating from different abstraction layers. This unified view enables accurate reconstruction of execution paths across serverless functions and facilitates effective correlation during attack analysis. To overcome the limita-

tions of intrusive logging mechanisms and version-dependent instrumentation, the thesis further introduces a runtime log collection and synchronization methodology that leverages modern in-kernel tracing capabilities. It achieves low-overhead, platform-agnostic logging while preserving causal consistency of log entries across distributed hosts using kernel-level vector clocks and synchronized buffer mechanisms.

Recognizing that real-world attack detection must operate under conditions of incomplete labeling and evolving threat behavior, the next contribution of the thesis advances the provenance analysis by enriching the provenance graph with semantic attributes, such as system call arguments, file/socket metadata, and process namespace features. It then formulates the anomaly detection problem as an unsupervised graph reconstruction task, wherein the model learns to represent normal system behavior through latent embeddings of heterogeneous entities and their interactions. Deviations from these representations are then interpreted as potential indicators of abnormal or malicious behavior. The approach is inherently robust to unseen or zero-day attacks and does not rely on prior knowledge of vulnerabilities or labeling of events.

The proposed methodology is rigorously evaluated through a series of real-world deployments on large-scale microservice benchmarks and controlled attack simulations using known CVEs. The results demonstrate marked improvements in attack detection accuracy, precision, and scalability over state-of-the-art baselines. Additionally, the thesis quantifies improvements in log fidelity, reduction in irrelevant log noise, and end-to-end detection latency, while maintaining a minimal resource footprint suitable for production-grade deployments.

In summary, this research contributes a principled and practical solution to the problem of secure observability and attack forensics in cloud-native environ-

ments. By integrating causality-aware logging, provenance modeling, and graph-based anomaly detection into a cohesive pipeline, it lays the groundwork for a new generation of security analytics systems tailored for the complexity and dynamism of distributed microservice architectures.

Keywords: System Observability, Runtime Provenance, Causality Tracking, eBPF-based Monitoring, Provenance Graph, Attack Detection, Dynamic Log Analysis, Log Correlation, Distributed Microservices

Contents

List of Figures

List of Tables

Nomenclature

Abbreviations

μS	Microservice
AIQL	Attack Investigation Query Language
ANN	Artificial Neural Network
APT	Advanced Persistent Threat
ASCL	Application-Specific Common Log
CFG	Control Flow Graph
CNCF	Cloud Native Computing Foundation
CoD	Computing-on-Demand
CVE	Common Vulnerabilities and Exposures
CWE	Common Weakness Enumerations
DAG	Directed Acyclic Graph
eBPF	Extended Berkeley Packet Filter
FD	File Descriptors
FPR	False Positive Rate
GAE	Graph Autoencoder
GAT	Graph Attention Network
GCN	Graph Convolutional Network
GNN	Graph Neural Network
HGAE	Heterogeneous Graph Autoencoder

HID Host Identifier

IIoT Industrial Internet of Things

IT Information Technology

KNN K-Nearest Neighbors

LKM Loadable Kernel Module

LMGF Log Message Generating Functions

LMS Log Message String

LMS-CFG Log Message String - Control Flow Graph

LOF Local Outlier Factor

LSM Linux Security Modules

ML Machine Learning

MTTR Mean Time To Repair/Mean Time To Recovery

OS Operating System

PCA Principal Component Analysis

PCB Process Control Block

PID Process Identifiers

PoC Proof-of-Concept

RCE Remote Code Execution

SaaS Software as a service

SLC serverless computing

SSRF Server-Side Request Forgery

SVM Support Vector Machine

TCB Trusted Computing Base

TID Thread Identifier

UPG Universal Provenance Graph

VM Virtual Machine

Symbols

α, β and γ Hyperparameters used to control the relative importance

$\mathbb{L}$ Contains the LMGF from $\mathcal{L}$

$\mathcal{L}$ List of standard LMGF names

μ_{train} The mean of the anomaly scores

σ The ReLU activation function

σ_{train} Standard deviation of the anomaly scores

θ Threshold

AGG Aggregate Function

BT Depth of backward tracing threshold

$E_{edge}(G)$ Edge Reconstruction Error

$E_{node}(G)$ The Node Reconstruction Error

$E_{struct}(G)$ Structure Reconstruction Error

$h_{\tau}^{(l)}$ Embeddings of type τ nodes at layer l

$W_{\tau,\tau',r}^{(l)}$ The learnable weight matrics for cross-type message passing

$W_{\tau}^{(l)}$ Self loop weights

N(τ The neighbour node types of τ

R(τ, τ') The relation (edge) types between the node type τ and τ';

Chapter 1

Introduction

The increasing adoption of microservice-based architectures and serverless computing platforms has transformed the design, deployment, and operation of modern distributed systems. Unlike traditional monolithic architectures [1], where the entire application is deployed as a single unit, microservices decompose functionalities into smaller, independently deployable services that communicate over a network [2]. While this architectural paradigm brings benefits in terms of scalability, modularity, and faster release cycles, it also introduces significant challenges in terms of system observability, security, and forensic analysis. The challenges that arise due to such infrastructure are increased complexity and a tendency to be more error-prone. It becomes harder to understand the system behavior as a whole, how different services interact, and why some errors occur. As the system can be spanned or spread across multiple machines, data centers, and virtual machines, it is difficult to understand and observe the system's flow of data and control. For monolithic applications deployed on a single machine, we can use the standardized logging tools (e.g., Auditd [3], Syslog [4], and Journald [5]) to understand the system events. However, these logs quickly lose value when the system becomes complex because they become hard to search and correlate.

Recent works [6, 7] have emphasized the growing attention toward the concept of observability in the current research landscape. Formally, Observability is defined as the ability to derive the internal state of a system from its external output [8]. Monitoring

detects if something is wrong, whereas observability helps investigate why something has gone wrong in complex distributed systems where problems may not be easily detectable. It observes the system's behavior or state without peeking under the system's hood. This helps developers to keep track of, understand, and fix issues in complex systems. It involves a set of tools to gather and analyze data collected from the system. Hence, logging can enable system observability if logs contain information relevant to the system behavior or state that is being observed.

Observability, being able to deduce the internal state of a system from its external outputs, is a basic requirement of large-scale distributed application security and management. Logging, system monitoring, and audit tools are the basis of traditional observability. In containerized and sandboxed environments, these tools fail because of issues such as *namespace isolation*, *asynchronous execution*, *broken logging*, and *lack of standard log formats* [9]. This leads to partial system visibility, making it difficult to trace events across services and hosts, detect anomalies, or investigate attack patterns effectively.

To overcome these limitations, the concept of provenance has gained traction in the systems and security communities. Where observability relies on understanding the origin and history of data, but to ensure data quality and reliability **Provenance** provides the required context for data observability by tracking data's source, transformations and usage, enabling better monitoring and root cause analysis [10]. Provenance graphs, which are the structured representations of system events, can be used to analyze causal relationships between processes, files, and network events. They serve as a powerful foundation for both real-time anomaly detection and retrospective attack investigation.

Hence, observability relies on provenance, and provenance informs observability. Basically, provenance provides the "what" and "how" of data, while observability provides the "why" and "when" of the data behaviour. The provenance data (logs in our case) can be transformed into a provenance graph to illustrate the interaction between the system's components. It represents the relationship between the control flow and data flow between system subjects (such as processes, threads, etc.) and system objects (such as files, sockets, and registry) in the system through a directed graph with timestamps.

Our research understands the core problem of enhancing system observability and

provenance tracking in distributed microservice environments. It proposes a suite of novel frameworks and methodologies that enable *unified, context-rich log collection* across distributed hosts and construct *dynamic, causally consistent provenance graphs*. These graphs are further analyzed using machine learning and graph-based methods for effective attack detection, particularly against sophisticated threats such as *Advanced Persistent Threats* (APTs) [11, 12]. The research contributions in this thesis span the design and implementation of logging frameworks, provenance graph construction algorithms, and anomaly detection models. We first design a log collection module to collect system-level provenance efficiently and accurately. It provides fast and efficient query interfaces. Next, the attack detection module processes a large amount of log files and the provenance graphs, and locates the suspicious behavior. Our framework is evaluated through synthetic benchmarks and real-world integrations, focusing on completeness and accuracy.

1.1 Motivation

The transition from monolithic to serverless and microservices has introduced operational complexity along with architectural flexibility [13, 14]. Microservices usually execute as containers on many hosts, and they communicate asynchronously with one another. The motivation for this work is to enhance the observability of distributed microservice systems, as the lack of unified observability can lead to several issues, such as investigating anomalies can lead to dead ends, investigation can be time-consuming, manual intervention to debug the issues, etc, which can degrade the performance of the application. Hence, an efficient observability tool can help in cost saving by faster incident resolution and reduced manual intervention. Here, we explore the motivation behind the problems addressed in this thesis by focusing on four significant aspects related to provenance tracking, observability, and attack detection.

1.1.1 Lack of unified provenance tracking in Serverless and Containerized Environments

Serverless and containerized (SLC) applications pose unique challenges for provenance tracking due to their distributed, ephemeral, and sandboxed nature. Existing system-level logging tools like `FUSE` [15], `LSM` [16], and `CamFlow` [17] require OS-level instrumentation, making them unsuitable for lightweight SLC deployments. Provenance analysis frameworks such as `SLEUTH` [18], `POIROT` [19], and `HOLMES` [20] struggle with the *dependency explosion* problem [21, 22] and are designed primarily for monolithic systems. While tools like `BEEP` [22], `LogGC` [23], and `ProTracer` [24] attempt to partition execution paths, they often require access to application internals. Recent works such as `OMEGALOG` [9] and `ALCHEMIST` [25] explore hybrid log correlation, but lack support for language-independence or non-invasive. Tools specific to SLC systems, such as `X-Trace` [26], `PivotTracing` [27], and commercial solutions like `IOPIPE` [28], `Dashbird` [29], `Epsagon` [30], `Thundra` [31], are either invasive or language-bound. This gap motivates the design of *DisProTrack*, a non-intrusive, language-agnostic framework capable of generating unified provenance graphs by combining system and application logs for root cause analysis in distributed SLC environments.

1.1.2 Need for a Causally-Consistent Observability Framework for Distributed Microservices

Existing logging solutions based on kernel instrumentation [9, 14], proxies [10, 32], or LKMs [33] are not scalable or robust, especially under kernel upgrades [34]. Prior provenance systems like `SLEUTH` [18] are optimized for monolithic or single-host settings and cannot track causality across containerized services. Commercial tools such as `Datadog` [35] and `Dynatrace` [36] offer only coarse observability and lack support for deep system-level context. Although **Extended Berkeley Packet Filter (eBPF)**[1] has shown promise in system observability [37–39], existing eBPF-based loggers like `Tracee` [40] fail to combine application and system logs while preserving causal order. This highlights the need for

[1] https://ebpf.io/what-is-ebpf/ (Last accessed: May 5, 2026)

a non-invasive, language-agnostic observability framework capable of capturing causally-consistent, multi-layer logs in distributed environments, leading to the design of *XPLOG*.

1.1.3 APT Investigation using Provenance Data in Distributed Microservices

Advanced Persistent Threats (APTs) [11, 12] in distributed microservice applications often exhibit stealthy, multi-stage behavior that spans containers, services, and hosts. Effective detection requires holistic analysis across both system and application layers. However, existing solutions such as `ATLAS` [41], `AIQL` [42], and `OmegaLog` [9] primarily rely on host-level logs or query-based investigation, lacking visibility into distributed causal interactions. Recent ML-based methods [43–46] leverage provenance graphs but fail to capture cross-service behavior due to isolated or coarse-grained graph representations. Additionally, container sandboxing and namespace isolation limit system call visibility, making cross-host correlation challenging. Traditional provenance tools [47, 48] and audit mechanisms like `Linux Audit` [3] are insufficient in capturing fine-grained, container-aware provenance. These gaps underscore the need for a framework like μ*Prov*, which unifies and correlates distributed logs from both system and application layers to generate a comprehensive runtime provenance graph for effective APT detection across microservices.

1.1.4 Enriched Provenance Graphs for Unsupervised APT Detection in Microservices

Although supervised models can achieve high accuracy with curated labeled data, their applicability is limited in real-world, dynamic environments where zero-day attacks and evolving threat patterns are common. In cybersecurity, labeled datasets are scarce due to the high cost and expertise required for annotation, and often fail to generalize across diverse system configurations and workloads. Moreover, supervised models are susceptible to overfitting, especially in distributed microservice environments with varied execution paths and inter-service dependencies. Recent graph-based anomaly detection methods such as `Prov2Vec` [49], `ThreatRACE` [50], `DeePro` [43], and `ProvDetector` [51]

leverage graph neural networks (GNNs) but assume homogeneous graph structures and lack support for semantically rich system entities like files, processes, and sockets. Crucially, these models overlook the distributed and context-sensitive nature of microservices, making them inadequate for detecting threats in containerized applications. These limitations motivate the development of μ*ProvGAE*, which builds semantically enriched, heterogeneous provenance graphs and leverages unsupervised graph autoencoders for detecting APTs across distributed microservices without labeled data or application instrumentation.

1.2 Objectives of the Thesis

As observed in the previous section, tracking provenance in distributed platforms presents several challenges. In this section, we outline the objectives of this thesis, which aim to address the identified research problems.

1.2.1 Unified Provenance Tracking for Distributed Serverless Microservices

Modern microservice-based applications operate in containerized or serverless environments, where logs are generated independently by loosely coupled services. These logs often differ in format, semantics, and granularity, making it difficult to perform system-wide event correlation or root-cause analysis. Existing logging solutions are either restricted to system-level data or require invasive instrumentation, lacking the ability to holistically connect application behavior with underlying system activity. The first objective of this thesis is to build a unified provenance tracing framework that constructs a Universal Provenance Graph (UPG) by seamlessly integrating application and system logs across microservices. This objective involves: (i) capturing heterogeneous log streams from distributed services without modifying application code, (ii) designing a static control-flow graph (CFG) extractor that maps application logs to their execution paths, and (iii) constructing a scalable and semantically consistent UPG that bridges the gap between system-level and application-level provenance.

1.2.2 Causally Consistent Observability in Sandboxed Distributed Systems

Achieving observability in sandboxed microservices is challenging due to isolation across containers, variability in execution environments, and fragmented log sources. Conventional tools either rely on intrusive instrumentation or fragile Loadable Kernel Modules (LKMs), which are not suitable for modern container orchestration systems. The second objective of this thesis is to design a platform-independent, grey-box observability framework that captures causally consistent and context-aware logs across distributed, containerized applications without requiring LKMs, application instrumentation, or kernel modifications. The framework is powered by eBPF, enabling low-overhead, fine-grained monitoring of system and application events. The sub-objectives include: (i) developing an eBPF-based log collector that operates seamlessly across sandboxed containers, (ii) preserving causal consistency and log atomicity using synchronized ring buffers, and (iii) evaluating the framework's robustness, scalability, and efficiency under real-world deployment settings against widely-used enterprise solutions.

1.2.3 Provenance Graph Construction for Attack Investigation

Detecting multi-stage attacks such as Advanced Persistent Threats (APTs) in distributed microservices is complicated by the lack of integrated observability and the absence of realistic benchmarking datasets. Traditional methods treat logs in silos and fail to capture the causal flow of complex attack sequences across services and layers. The third objective of this thesis is to develop a dynamic provenance graph-based framework for attack investigation, enabling system-wide tracing of event lineage to uncover APT behaviors. This includes: (i) extracting detailed provenance graphs from kernel-level system calls in a non-intrusive manner, (ii) capturing causal relationships among processes, files, and network events to expose anomalous patterns, and (iii) constructing a hybrid dataset combining real-world application logs with emulated attacks based on known CVEs/CWEs to facilitate empirical evaluation and accuracy-performance analysis.

1.2.4 Unsupervised Anomaly Detection Using Enriched Provenance Graphs

Existing anomaly detection approaches rely heavily on labeled data and often lack semantic context, limiting their ability to detect evolving or zero-day attacks in dynamic microservice environments. The fourth objective of this thesis is to develop an unsupervised anomaly detection framework using enriched heterogeneous provenance graphs that model both the structure and semantics of system behavior. This involves: (i) enhancing provenance graph construction by embedding rich contextual attributes (e.g., file modes, socket metadata, process namespaces) using an advanced eBPF logging backend, (ii) designing a tailored Heterogeneous Graph Autoencoder (GAE) that learns latent representations of normal system behavior across diverse node and edge types, and (iii) detecting anomalies by measuring graph reconstruction errors without requiring labeled attack data, thereby ensuring adaptability to unseen and evolving threats in real-world deployments.

1.3 Contributions of the Thesis

In the following, we outline how each research objective has been addressed through the specific contributions of this thesis.

1.3.1 Distributed Provenance Tracking over Serverless Applications

The first contribution of the thesis develops a provenance tracking system called *DisProTrack*, designed for distributed serverless computing (SLC) platforms. Unlike prior works which focus on monolithic applications running directly over the kernel, *DisProTrack* addresses the key challenges posed by serverless architectures namely, the asynchronous generation of heterogeneous system and application logs, shared PID spaces, and difficulty in mapping control flows across execution units. *DisProTrack* introduces the notion of a Universal Provenance Graph (UPG), which combines application logs and system logs into a unified, queryable structure. The core contributions of *DisProTrack* are as follows. First,

we implement a static analyzer that generates Log Message String-Control Flow Graphs (LMS-CFGs) from application binaries. These graphs encode the relationships among log-generating functions and are later used to contextualize runtime log messages. Second, we design a Loadable Kernel Module (LKM) that intercepts system calls at runtime, tags them with container ID, PID, and timestamp, and correlates them with LMS entries to extract meaningful execution units. Third, we design an algorithm to construct the UPG by mapping execution units defined as bounded sequences of LMSes and corresponding system calls onto graph components. Nodes in the UPG represent entities such as executables, files, and sockets, while edges capture the syscalls that induce causal relationships. By converting LMSes into regular expressions, *DisProTrack* reduces graph size, minimizes false matches during query processing, and mitigates dependency explosion. Finally, we evaluate *DisProTrack* using seven benchmark SLC applications (e.g., MySQL, Apache, PHP-FPM, DynamoDB, ImageMagick, OpenVPN, wget). Static analysis shows that back-trace depth (BT) affects LMS discovery, memory consumption, and analysis time. For instance, MySQL required the highest memory ($\sim 1000MB$) and time ($\sim 1000s$), while PHP-FPM produced the highest number of LMSes despite a smaller size. Runtime evaluation is performed on three web-based scenarios (E1: registration-login, E2: password reset, E3: malicious script injection). The runtime engine generates and parses ASCLs in under 1s, with total provenance graph construction time under 30s and memory footprint < 250B. Scenario E3 produces larger UPGs (more nodes and connected components), reflecting more complex behaviors, including a simulated confidential data theft attack. *DisProTrack* successfully traces the deviation caused by the mal.sh script, capturing its access to */etc/*release* and data exfiltration via socket connections. Overall, *DisProTrack* provides an efficient, deployable, and explainable provenance tracking solution for container-based serverless platforms. The implementation is open-sourced and demonstrates the feasibility of attack investigation via causality-preserving log correlation without application instrumentation. However, the current method is limited to the deployability as it leverages the Linux Kernel Module (LKM). In the following research work we developed a system leveraging eBPF to overcome the deployability issues associated with LKM that has a significant dependency on the Linux kernel versions.

1.3.2 A Dynamic Observability Framework for Distributed Sandboxed Microservices

In the next contributory chapter, we design a distributed, cross-layer provenance logging framework named *XPLOG* to address the challenges of preserving execution context and causality in cloud-native microservices. *XPLOG* is designed to be lightweight, scalable, and capable of generating real-time causally ordered logs by combining system-level and application-level events across containerized environments. We design a novel grey-box observability model that operates without any application instrumentation. At its core, *XPLOG* consists of two major components: *XPLOG* Agent and *XPLOG* Collector that work in tandem to monitor activities on each host and aggregate provenance-rich logs centrally for forensic analysis and anomaly detection. *XPLOG*'s Agent runs as privileged agents on each host, intercepting syscall events and enriching log entries with contextual metadata like process/thread IDs, syscall arguments, and executable paths. These logs are collated using a host-specific eBPF ring buffer and are guaranteed to be causally ordered due to our atomic syscall tracing strategy. The key contributions of *XPLOG* are as follows. First, we implement a userspace agent using eBPF that selectively traces 19 system calls across file, socket, and process domains, while simultaneously capturing structured application logs. *XPLOG* leverages vector clocks to maintain causality across threads and containers within and across hosts. Application logs are intercepted at runtime by hooking into write syscalls, mapping them to process/thread IDs, and associating them with semantic context. All traced logs are passed through a causal filter and forwarded periodically to the collector. Second, the *XPLOG* Collector correlates incoming logs from multiple hosts and constructs a Unified Provenance Log, preserving inter-host event ordering. It performs dynamic correlation between application and system call logs using shared fields (e.g., Host PID, Host TID) and ensures consistency of distributed execution traces. A causality-aware filtering mechanism reduces noise and avoids redundant entries while maintaining full observability. Third, *XPLOG* supports dynamic reconfiguration of logging granularity and filtering based on runtime context and application needs. This flexibility allows the framework to scale across heterogeneous workloads while maintaining low overhead. The logs are emitted in a standardized JSON format and exported to a centralized store, enabling external queries. Third, we conduct a comprehensive evaluation using the DeathStarBench benchmark suite deployed over 30 microservices across 10 to 30 hosts with workloads of 10K, 20K, and

30K requests. We show that it significantly reduces log disorder (up to $8\times$ compared to Tracee) and improves the richness of collected logs, capturing both syscall and application contexts. XPLOG maintains low system overhead, consuming < 3.2% CPU and ~ $1.7Mbps$ network traffic while handling up to 30K concurrent requests. It generates ~ 2 to 6 million log entries for 30K requests. Each log entry is ~ 500 to 600 bytes. We observe a memory footprint of ~ $256KB$ per host with ring buffers supporting ~ 437 entries. Despite a 50 ms polling interval, *XPLOG* exhibits < 6% buffer usage, indicating robustness under high load. Further, the per-request latency overhead remains negligible (< 1%), and agent CPU utilization is consistently below 5% under peak load. Additionally, we demonstrate that XPLOG enables rapid, relevant log retrieval using event-sequence-based filters with zero irrelevant or missed logs, in contrast to Tracee, which misses relevant entries and returns many unrelated logs. Ablation studies show the importance of maintaining atomic syscall ordering and distinguishing container contexts using PID namespaces. Finally, we present qualitative results comparing *XPLOG* and Tracee, showing that *XPLOG* logs are more expressive, easier to interpret, and provide vital details like request IDs, socket addresses, and file paths, making them more suitable for forensic analysis and dynamic provenance tracking in real-world, distributed microservice deployments. In the next work, we explore how these enriched causally-ordered logs can be transformed into dynamic provenance graphs for attack detection using classical supervised machine learning models.

1.3.3 Supervised Attack Investigation in Microservices Using Provenance Graphs

Next, we design a provenance-based system called *μProv* to enable attack investigation across distributed microservice environments. At its core, *μProv* constructs dynamic and scalable runtime provenance graphs that correlate system-level and application-level activities across multiple hosts using eBPF instrumentation. The main contributions of *μProv* are as follows. First, we utilized our prior work *XPLOG*, a custom eBPF-based logging solution that operates at the host OS kernel level to capture causally ordered logs from microservices deployed in containers across multiple hosts. This logging framework ensures causal consistency using kernel-space vector clocks and collects enriched logs across system and application layers. These logs capture task context, arguments, and artifact

metadata and are streamed to a central collector, enabling real-time provenance analysis. Second, we design a dynamic provenance graph generator that incrementally constructs directed acyclic graphs (DAGs) from streaming logs. Each node in the graph represents processes, files, or sockets, while edges represent system calls and their temporal dependencies. The graph captures multi-host causal paths and distinguishes between normal and anomalous behavior by preserving system-wide execution semantics. The provenance graph construction is based on an algorithm that maps logs to nodes and edges using 21 critical system calls across file I/O, process, and socket domains. Third, to support realistic attack detection, we develop PicShare, a dockerized microservice-based photo-sharing application that integrates CVE- and CWE-based vulnerabilities (e.g., SSTI, RCE, SQL injection). This enables the emulation of multi-stage APTs and the generation of labeled datasets for training attack detection models. We emulate benign and malicious scenarios across 13 microservices deployed on 10 VMs using Docker Swarm. Finally, we conduct an extensive evaluation to compare *μProv* with Tracee, an existing eBPF-based tracing tool. We use supervised classification models (KNN, SVM, Random Forest, ANN) trained on graph-based and provenance-derived features. Results show *μProv* achieves up to 87.78% accuracy (vs. 52.80% for Tracee), reduces false positives and false negatives, and detects attacks $\sim 50\%$ faster (5.2s vs. 10.2s). *μProv* also incurs lower resource overhead (10MB memory vs. 200MB for Tracee) and generates more granular provenance graphs (higher nodes and edge counts per event), enabling a more comprehensive view of attack sequences. Overall, *μProv* provides a robust and lightweight provenance framework tailored to distributed microservice architectures for effective and scalable attack investigation. In the next work, we extend this work to incorporate advanced graph learning techniques over the generated provenance graphs for unsupervised anomaly detection in evolving microservice environments

1.3.4 Graph-based Unsupervised Learning for System Provenance

In the final contribution of the thesis, we design an advanced attack detection system named *μProvGAE* that constructs enriched heterogeneous provenance graphs for unsupervised anomaly detection in distributed microservice environments. *μProvGAE* builds upon our prior framework, *μProv*, which used supervised learning for attack detection from system-level logs. The primary motivation stems from the limitations of prior approaches: existing

provenance systems treat logs in isolation, ignore semantic context, or rely heavily on labeled data, which is scarce in real-world APT scenarios. *μProvGAE* addresses these challenges through three key contributions. First, we utilized our prior developed eBPF-based logging framework, *XPLOG* that captures causally ordered, fine-grained system and application logs across distributed containers. These logs preserve semantic information like process context, file access flags, and network socket metadata, enabling the construction of expressive provenance graphs. Second, we design a heterogeneous graph autoencoder (HGAE) that learns normal behavior patterns from these graphs. It models heterogeneous nodes (process, file, socket) and diverse system call-based edge types, and reconstructs node/edge features and structure in an unsupervised manner. Third, anomaly detection is done by computing reconstruction errors at the graph level. Graphs with higher reconstruction error are flagged as anomalies, allowing detection of evolving or zero-day APT attacks. We evaluate *μProvGAE* using a realistic, containerized photo-sharing application (PicShare), where we emulate CVE-based attack scenarios including SQL injection, insecure file uploads, and server-side template injection. *μProvGAE* achieves 85.27% F1-score, outperforming both supervised (UNICORN: 83.70%) and unsupervised (ProvDetector: 47.57%) baselines on enriched provenance graphs. Ablation studies show the importance of node, edge, and structural loss components. Further, *μProvGAE* improves detection precision (95.61%) and recall (69.16%) over traditional ML classifiers trained on *μProv* logs (e.g., Random Forest with 87.78% accuracy), and outperforms Tracee (52.80%) by leveraging better semantic context. Finally, the enriched provenance graph construction and unsupervised anomaly detection pipeline are evaluated on 10 VMs using Docker Swarm, with 1000 requests per scenario. *μProvGAE* demonstrates practical viability for real-time APT detection across microservices, marking a shift from host-level coarse models to semantic-aware, causally consistent graph-based observability.

In summary, this thesis introduces a unified suite of systems for provenance tracking and attack detection in distributed environments. *DisProTrack* constructs provenance graphs in serverless platforms using kernel-level analysis, while *XPLOG* leverages eBPF to capture causally ordered logs across microservices. Building on these, *μProv* enables supervised attack detection via dynamic provenance graphs, and *μProvGAE* extends this to unsupervised anomaly detection. Collectively, these systems offer scalable, explainable, and semantically rich observability for cloud-native security.

1.4 Organization of the Thesis

The rest of the thesis is organized as follows.

- **Chapter 2** presents the background and a comprehensive literature survey on provenance tracking, system observability, and APT detection in distributed sandboxed environments. It identifies key research gaps and limitations, organized into four subdomains aligned with the thesis objectives, and outlines how this work addresses those gaps.
- **Chapter 3** presents *DisProTrack*, which introduces a provenance tracking framework to generate a Universal Provenance Graph (UPG) from a universal log file (combined logs from system and application logs) and also uses Loadable Kernel Module (LKM) for runtime unit identification over the logs by intercepting the system calls with the help from the control flow graphs over the static application binaries.
- **Chapter 4** presents *XPLOG*, which develops a scalable, pluggable, easily deployable, and dynamic runtime observability framework for distributed sandboxed computing platforms that leverages the capability of 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.
- **Chapter 5** presents *μProv*, which provides a framework for generating robust, scalable, and dynamic provenance graphs to aid in attack investigation over distributed microservice architectures using classical supervised machine learning models.
- **Chapter 6** presents *μProvGAE*, an extended version of *μProv* for detecting APT attacks in a distributed microservice architecture, while utilizing an unsupervised model leveraging heterogeneous Graph Autoencoders (GAEs) to improve the anomaly detection accuracy by constructing enriched provenance graphs.
- **Chapter 7** concludes the thesis by summarizing our contributions briefly and highlighting the possible future directions.

Chapter 2

Background & Related Work

This chapter presents the background and surveys existing literature across the areas of system-level logging, provenance tracking, causality analysis, and attack detection, with a particular emphasis on distributed, containerized microservice environments.

2.1 Background

To ensure secure and explainable behavior in distributed microservice systems, it is crucial to maintain an in-depth understanding of the system's runtime execution. This requires capturing not just the events themselves but also the semantic, temporal, and causal relationships among them. Our work builds upon three fundamental concepts: **provenance**, **causality**, and **observability**, which collectively form the foundation of dynamic analysis and attack investigation frameworks.

2.1.1 Provenance Tracking

Data provenance represents the comprehensive tracking and documentation of data origins, transformations, and lineage throughout its lifecycle in computing environments. In

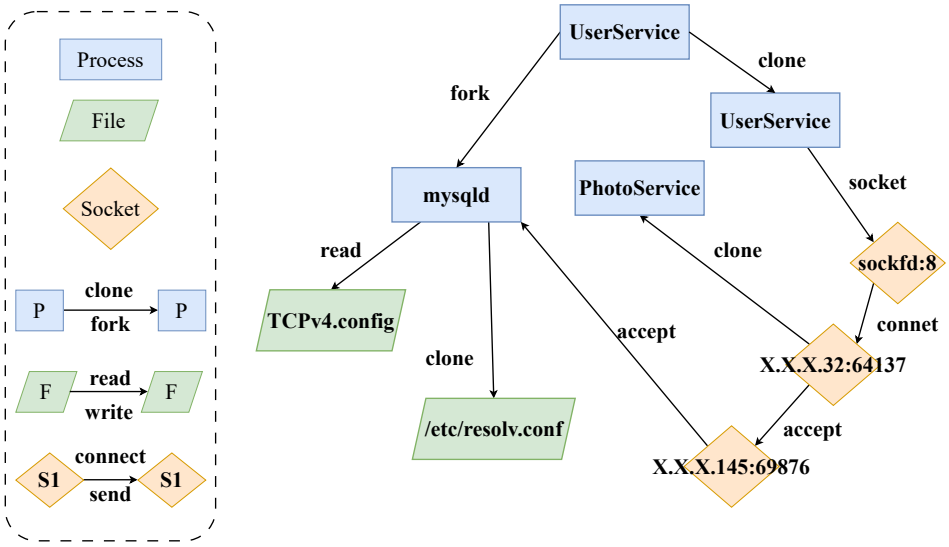


Figure 2.1 Example of a Provenance Graph (Rectangle: processes; Parallelogram: files; Rhombus: Socket/Network Connection). UserService/PhotoService are the microserives running on a host.

distributed systems, provenance refers to the metadata that records the entities, users, and processes involved in the history and evolution of data objects, ensuring data integrity, reliability, and transparency crucial for debugging, auditing, and compliance purposes. Provenance metadata encompasses both file operations and derivation history, serving as essential information to evaluate data quality and understand system behavior.

The theoretical foundation of provenance systems rests on directed acyclic graphs (DAGs) where G = $\langle V, E \rangle$, with vertices V representing system entities (processes, files, network sockets) and edges E capturing causal relationships between them (shown in Figure 2.1). System-level data provenance describes information flow between system entities by analyzing system calls in user or kernel space, creating structured representations that enable forensic analysis and attack investigation.

The provenance framework must capture and maintain multiple dimensions of metadata throughout the data transformation pipeline (as shown in Figure 2.2). These dimensions include version control information (determining which specific version of input files are being utilized), environmental context (identifying the operating system and runtime environment), dependency tracking (cataloging the functions, libraries, and external

dependencies involved in the processing), system state monitoring (documenting background processes and concurrent system activities), data lineage mapping (tracing how data flows and merges from various sources into the final output), workflow visualization (representing the sequential processing steps and their interdependencies), and information granularity control (determining the appropriate level of provenance detail to capture and display). This multi-faceted approach to provenance collection ensures that the final artifact maintains complete traceability back to its origins while preserving sufficient contextual information to enable reproducibility, debugging, and trustworthiness assessment. The diagram effectively demonstrates why modern provenance systems require sophisticated metadata management capabilities to address the complexity of contemporary computational workflows, particularly in distributed and microservice-based environments where multiple systems, versions, and dependencies interact to produce final outcomes.

The scalability challenge in provenance systems stems from the performance overhead and storage requirements needed to capture comprehensive metadata in large-scale, data-intensive computing environments. Traditional approaches like SPADE (Support for Provenance Auditing in Distributed Environments) [52] utilize graph databases for storing audited provenance data, while systems such as FusionFS [53] implement distributed file metadata management using distributed hash tables. However, these approaches often require kernel instrumentation or application code modification, making them impractical for dynamic, containerized environments.

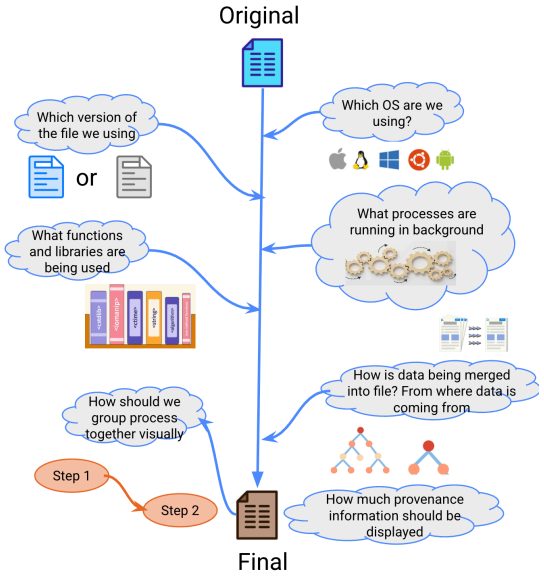


Figure 2.2 Examples of Metadata being Captured during System Provenance

Provenance-based intrusion detection systems (PIDS) have emerged as a critical security application, leveraging provenance graphs to detect advanced persistent threats (APTs). Research initially focused on data collection mechanisms, then progressed to graph summarization and storage optimization, with recent emphasis on automated in-

trusion detection capabilities. The challenge lies in achieving **causally-consistent provenance collection** across distributed nodes while maintaining system performance and avoiding the dependency explosion problem. In cloud computing contexts, provenance data operates across multiple architectural layers - infrastructure, virtualization, application, platform, and client tiers - with each layer addressing different stakeholder requirements. Cloud providers utilize infrastructure provenance for resource utilization auditing, while application developers require fine-grained execution provenance for debugging and performance optimization.

2.1.2 Causality

Causality in distributed systems establishes the fundamental ordering relationships between events, ensuring that cause-and-effect dependencies are preserved across multiple nodes despite the absence of global time synchronization. Leslie Lamport's seminal work "Time, Clocks, and the Ordering of Events in a Distributed System" defined the happens-before relationship "→" with three core principles: (1) if events a and b occur in the same process with a preceding b, then $a \rightarrow b$; (2) if a represents message sending and b represents message receipt, then $a \rightarrow b$; (3) transitivity: if $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$. The primary mechanism for maintaining causality is through logical clocks, which assign timestamps to events based on causal relationships rather than physical time. Lamport clocks provide total event ordering through scalar timestamps, where each node maintains a counter that increments with internal events and updates to max(local_clock, received_timestamp) + 1 upon message receipt. However, Lamport clocks cannot identify concurrent events that are causally unrelated, as the relationship $C(x) < C(y)$ does not necessarily imply $x \rightarrow y$. Vector clocks address this limitation by maintaining a vector $V[1...N]$ where $V[i]$ represents the logical time knowledge of process i. Vector clocks enable partial ordering of events and can reliably detect causality violations by comparing timestamp vectors element-wise. The formal definition establishes that $VC(x) < VC(y)$ if and only if $\forall z[VC(x)\_z \leq VC(y)\_z] \wedge \exists z'[VC(x)\_z' < VC(y)\_z']$, providing a mathematical foundation for causality detection. Vector clock operations follow specific update rules: local events increment the process's own clock component, message sending includes the current vector timestamp, and message reception involves element-wise maximum com-

putation followed by local increment. However, the scalability challenge of vector clocks lies in their O(N) space complexity per process. Causal consistency models in distributed databases ensure that causally related operations appear in the same order across all nodes.

2.1.3 Observability

Observability is a foundational concept in modern software engineering that refers to the ability to understand the internal state and behavior of a system based solely on the data it emits, such as logs, metrics, traces, and events. As distributed systems, microservices, and cloud-native architectures become more prevalent, observability has emerged as a critical practice for ensuring the reliability, performance, and maintainability of complex software environments. The primary goal of observability is to answer the question: "Why is the system behaving this way?", especially when unexpected issues occur. Rather than relying on predefined failure conditions or static dashboards, observability allows engineers to explore system behavior in real-time, uncovering root causes for both known and unknown problems (refer Figure 2.3).

At its core, observability is rooted in control theory, where a system is considered "observable" if its internal state can be determined through its external outputs. In practical terms, this means that a well-instrumented system should provide sufficient telemetry data to help operators infer what is happening inside, even when direct access to internal components is not feasible. The three pillars of observability, logs, metrics, and traces, play a vital role in this process. Logs are timestamped records of discrete events, useful for debugging and auditing. Metrics are numeric measurements that provide real-time insights into system health and performance, such as CPU usage, memory consumption, or request latency. Traces capture the end-to-end flow of requests across services, allowing teams to pinpoint bottlenecks and latency issues through distributed tracing.

In distributed microservices environments, observability becomes particularly challenging but also more valuable. Services often span multiple nodes, run asynchronously, and communicate over network protocols that can introduce delay or failure. Without observability, pinpointing issues in such systems is akin to finding a needle in a haystack. Observability tools, however, make it possible to correlate logs, metrics, and traces across

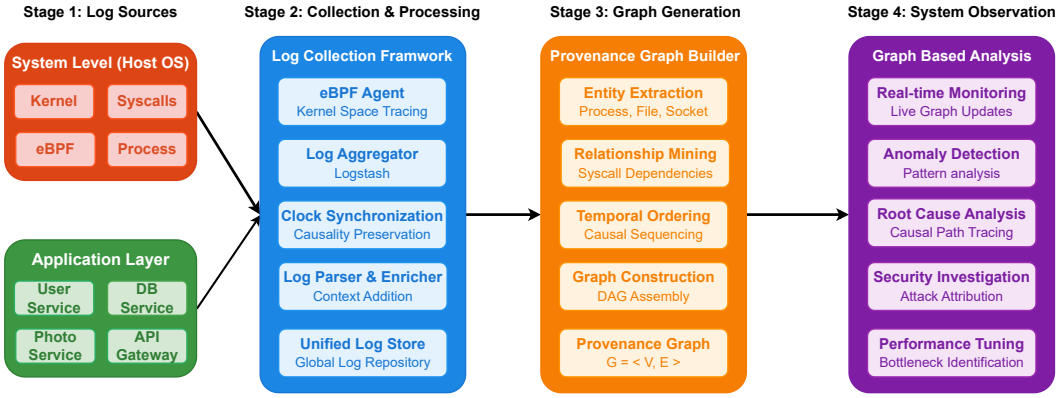


Figure 2.3 This diagram illustrates a comprehensive four-stage observability pipeline that transforms distributed system logs into actionable insights through provenance graph generation.

services and infrastructure, providing a cohesive view of how a request travels through the system. This holistic perspective enables rapid root cause analysis, reduces mean time to recovery (MTTR), and enhances operational efficiency.

Importantly, observability is not just about data collection; it is about actionable insight. A system can generate terabytes of telemetry data, but unless that data is structured, queryable, and contextualized, it offers little value. Modern observability platforms, therefore, incorporate features like log aggregation, real-time analytics, anomaly detection, and visualization dashboards to help engineers make sense of complex data streams. Moreover, observability goes hand-in-hand with DevOps and SRE (Site Reliability Engineering) practices, supporting continuous delivery, incident response, and service level objective (SLO) tracking.

In summary, observability empowers organizations to operate distributed systems with confidence. By transforming raw telemetry data into meaningful insights, it facilitates proactive issue detection, accelerates debugging workflows, and supports system resilience. As software systems continue to grow in scale and complexity, building observable architectures is no longer optional; it is a necessity for delivering reliable, performant, and secure services in dynamic production environments. The integration of artificial intelligence and machine learning into observability platforms enables automated anomaly detection, predictive maintenance, and intelligent alerting. Modern distributed systems leverage AI for

intelligent resource management, automated optimization, and pattern recognition in observability data streams, transitioning from reactive monitoring to proactive system health management.

2.1.4 Extended Berkeley Packet Filter (eBPF)

Extended Berkeley Packet Filter (eBPF) is a revolutionary technology that enables safe execution of sandboxed programs directly within the Linux kernel, fundamentally transforming system observability, networking, and security without requiring kernel modifications or risky kernel modules. Evolved from the original Berkeley Packet Filter designed for network packet filtering, eBPF now serves as a comprehensive kernel programming framework that allows developers to extend operating system functionality dynamically while maintaining system stability and security through rigorous verification mechanisms.

Traditional observability approaches suffer from significant limitations including high resource consumption from user-space agents, context-switching overhead, limited kernel visibility, and inability to capture real-time system events in containerized environments. eBPF addresses these challenges by operating directly in kernel space, providing near-zero overhead monitoring, comprehensive system call interception, and microsecond-precision event capture without missing critical system interactions that occur at the kernel level.

The eBPF architecture consists of several key components working in concert (refer Figure 2.4). eBPF Programs are custom code written in C or Rust that define the logic for data collection, filtering, or processing. The LLVM Compiler transforms source code into eBPF bytecode, which is then processed by the eBPF Verifier that ensures memory safety, prevents infinite loops, and guarantees program termination before execution. The Just-In-Time (JIT) Compiler optimizes bytecode for efficient execution, while eBPF Maps provide data structures for sharing information between kernel and user space. Hook Points (tracepoints, kprobes, uprobes, network interfaces) define where programs attach to intercept kernel events, and Helper Functions provide safe APIs for accessing kernel functionality and data structures.

eBPF excels in system observability by capturing comprehensive telemetry including

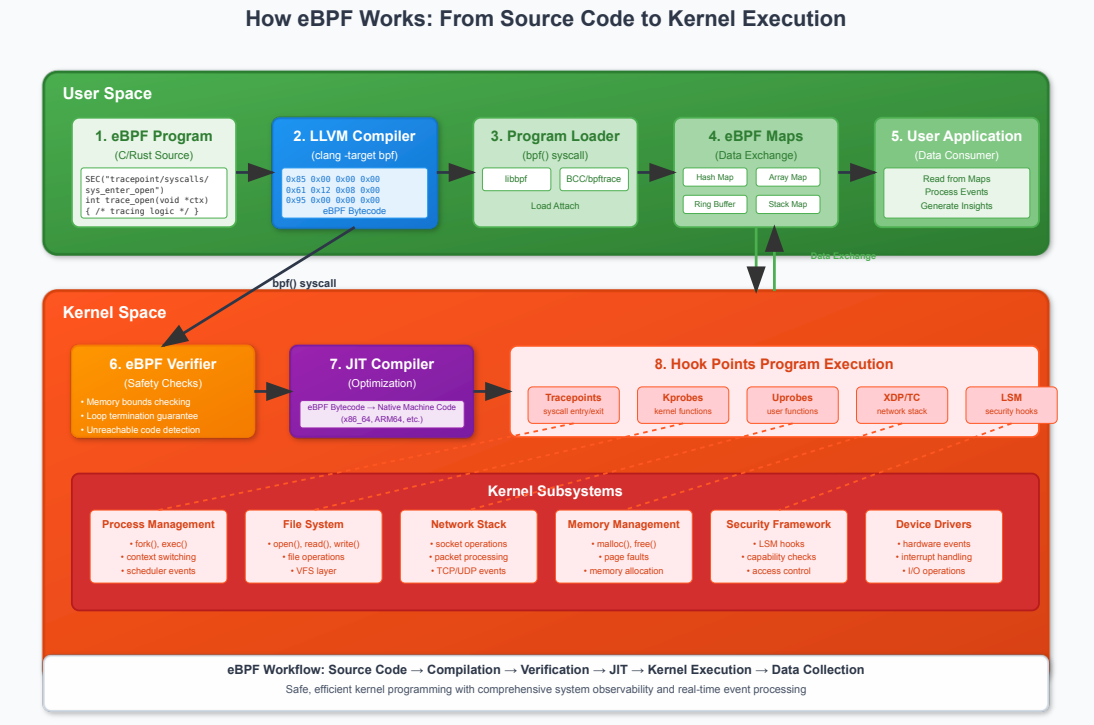


Figure 2.4 The eBPF Architecture showing the complete workflow from source code compilation, kernel verification, JIT optimization, and hook-based execution across kernel subsystems, with bidirectional data exchange via eBPF maps to enable safe, high-performance system observability and programmable kernel extensions.

process lifecycle events, file system operations, network socket activities, and system call arguments, enabling the construction of detailed provenance graphs for causal analysis and security investigation. In networking, eBPF enables high-performance packet processing, traffic shaping, and load balancing directly within the kernel, while security applications leverage eBPF for real-time threat detection, behavioral analysis, and policy enforcement. The technology's kernel-native approach provides unprecedented visibility into distributed microservice architectures, supporting advanced use cases including container network observability, performance profiling, and attack attribution through fine-grained system interaction tracking.

2.2 Related Work

The dynamic and ephemeral nature of microservices has challenged traditional approaches to system monitoring and threat detection, necessitating novel solutions rooted in provenance and causality. Accordingly, we survey the existing literature across four core themes:

1. Provenance Collection and Causality in Distributed Systems
2. Observability Infrastructure in Microservices
3. Provenance-based Threat Detection and APT Analysis
4. Graph-based Learning over Provenance for Anomaly Detection

2.2.1 Provenance Collection and Causality in Distributed Systems

To enable accurate forensic analysis and threat detection in cloud-native microservice environments, it is essential to capture detailed system-level provenance data and establish robust causal relationships across distributed components. This section surveys research efforts aimed at (i) system-level provenance collection, (ii) mitigating dependency explosion through execution partitioning, and (iii) integrating multiple layers of logs to improve causal reasoning and attack traceability.

(i) System-Level Logging and Provenance Collection

System-level provenance collection is foundational for constructing execution traces that capture causal links between processes, files, sockets, and system calls. Early tools such as FUSE [15] pioneered kernel-level hooking to track fine-grained file system operations. LSM [16] and Hi-Fi [54] leveraged the Linux Security Module framework to collect comprehensive whole-system provenance. CamFlow [17] introduced a modular provenance capture system, but it lacks support for distributed microservices. SPADE [52] utilized audit logs and graph databases to store system events, offering early foundations for attack re-

construction. ETW [55] focused on Windows environments by using event tracing for performance and forensic logging. Sleuth [18] implemented an in-memory dependency graph to enable real-time attack impact analysis. Pagoda [56] proposed efficient log ingestion and graph construction but assumed monolithic deployments. Poirot [57] and HOLMES [20] introduced semantics-based correlation techniques to detect multi-step attack campaigns over system audit trails. Watson [10] employed inter-process communication graphs to improve information flow tracking, although it struggled with containerized execution models. These systems laid the groundwork for provenance-based security, yet most are tailored for traditional, single-host setups and require significant kernel instrumentation, limiting their adaptability in dynamic containerized environments.

Table 2.1 Comparison of System-Level Provenance Collection Tools

Reference	System-Level Provenance	Distributed Support	Container Support	Kernel Instrumentation Required	Use of Audit Logs
FUSE [15]	✔	✗	✗	✔	✗
LSM [16]	✔	✗	✗	✔	✗
Hi-Fi [54]	✔	✗	✗	✔	✗
CamFlow [17]	✔	✗	✗	✔	✗
SPADE [52]	✔	✗	✗	✗	✔
ETW [55]	✔	✗	✗	✗	✔
SLEUTH [18]	✔	✗	✗	✗	✔
Pagoda [56]	✔	✗	✗	✗	✔
POIROT [57]	✔	✗	✗	✗	✔
HOLMES [20]	✔	✗	✗	✗	✔
WATSON [10]	✔	✗	✗	✗	✔

(ii) Dependency Explosion and Execution Partitioning

A persistent challenge in provenance-based analysis is the "dependency explosion" problem, where taint propagation or graph traversal leads to a combinatorial growth in dependencies, diluting signal quality and overwhelming analysts. Several works have addressed this by applying graph reduction and execution partitioning techniques. BEEP [22] attempted to control over-tainting by applying selective pruning of provenance edges during graph traversal. MPI [21] introduced a multi-path inference strategy to filter irrelevant causal links by clustering logically related execution paths. OmegaLog [9] improved semantic edge attribution using control flow integrity and log coalescing. ProTracer [24] performed selective tracing to capture relevant paths with minimal overhead. LogGC [23] pro-

posed log garbage collection for auditing systems to reduce trace volume while maintaining semantic integrity. ETW [55] also incorporated event correlation heuristics but remained platform-dependent. UIScope [58] enabled instrumentation-free partitioning by leveraging user-level observations and threading behavior. PivotTracing [27] and X-trace [26] allowed dynamic query-based tracing across distributed systems, although these techniques often required application-level modifications or lacked kernel context. Together, these methods address provenance scalability by reducing noise and isolating attack-relevant paths, but their dependency on source code instrumentation or runtime hooks limits applicability to modern, ephemeral microservices.

(iii) Multi-layer Log Integration for Causality Preservation

To enhance the granularity and accuracy of causality tracking, recent approaches integrate logs from multiple layers—application, middleware, and operating system. This holistic view allows better correlation of events and reconstruction of full request flows. Alchemist [25] proposed a framework to align application logs with system-level events, enabling more accurate behavioral modeling. OmegaLog [9], beyond its edge attribution methods, also supports fine-grained alignment of multi-source logs to establish causal paths across microservices. UIScope [58] avoids source-code instrumentation by relying on UI-level interactions to infer causal sequences. X-trace [26] introduced causal metadata propagation across services using user-level APIs, but lacks deep system-level insights. PivotTracing [27] allowed dynamic instrumentation for log correlation in distributed systems, though its reliance on JVM limits general applicability. Commercial observability tools such as IOPipe [28], Dashbird [29], Epsagon [30], and Thundra [31] offer fine-grained tracing for serverless platforms, but often depend on language-specific SDKs or require cloud vendor integration, which hampers portability. While these systems move towards end-to-end visibility, few provide seamless, language-agnostic causality support in fully distributed, multi-host, containerized environments.

Limitations and Scope

Despite significant progress in provenance collection and execution partitioning, major gaps remain. Many frameworks assume a monolithic or static environment and are incompatible with highly dynamic microservices. Most tools require invasive instrumentation or root access, restricting their deployment flexibility. Moreover, while multi-layer log fusion improves visibility, current systems lack generalizability and causal guarantees across hosts. These limitations highlight the need for a lightweight, non-intrusive, and scalable logging solution that can deliver causally consistent provenance graphs across distributed microservice systems.

2.2.2 Observability Infrastructure in Microservices

Effective observability in microservice systems is essential to capture runtime behaviors, understand system interactions, and enable root cause analysis. This section reviews advancements in (i) logging architectures, (ii) provenance tracking approaches in cloud settings, (iii) causality-based analysis tools, (iv) commercial observability platforms, and (v) eBPF-based monitoring, focusing on the limitations of existing systems in addressing the scale, heterogeneity, and dynamism of containerized infrastructures.

Table 2.2 Recent research work for logging frameworks compared to *XPLOG*

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
Horus [59]	2021	✗	✗	✓	✗	✓	✗	✓	✗	✓
Liu et al. [38]	2020	✓	✓	✓	✓	✗	✗	✗	✗	✓
Viperprobe [60]	2020	✓	✗	✓	✓	✗	✗	✗	✗	✓
Falcon [61]	2018	✗	✗	✓	✗	✓	✗	✓	✗	✗
SysFlow [62]	2020	✓	✗	✗	✓	✗	✗	✗	✗	✓
eAudit [63]	2023	✓	✗	✗	✓	✗	✗	✗	✗	✗
Sysdig [64]	2016	✓	✗	✗	✓	✗	✗	✗	✗	✓

(i) Logging in Distributed Cloud Infrastructure

Modern logging architectures aim to collect detailed telemetry across multiple layers of distributed systems. LogLens [65] and VNetTracer [66] provide real-time log analysis using language models and neural techniques for anomaly detection. ELT [67] extracts

structured features for anomaly detection from system logs. OMmegaLog [9] and ReverseMap [14] apply static and dynamic techniques for provenance generation and attack detection. Watson [10] and Alastor [32] design proxy-based and kernel-level modules to correlate audit logs with application-level activities. SLEUTH [18] constructs a real-time dependency graph from audit logs to aid attack visualization. DisProTrack [33] offers a pluggable logging system for containerized environments using kernel modules and eBPF hooks. LogGC [23] and LProf [68] apply static analysis and coarse-grained summarization for scalability, albeit at the cost of semantic precision. These systems demonstrate significant engineering innovation, but most rely on static kernel modules, do not support real-time graph generation, or lack generalizability across microservice topologies.

(ii) Provenance Tracking in Cloud

In cloud environments, provenance tracking has evolved from static DAG construction to real-time lineage monitoring. Hi-Fi [54] and CamFlow [17] use LSM-based hooks and kernel support for provenance capture. Watson [10] uses IPC graphs and control-flow tracing for taint propagation. SmartProvenance [69] explores privacy-preserving provenance using blockchain and smart contracts. Scippa [70] targets Android IPC-based attack detection using OS-level provenance. PROV2R [71] uses record-replay and taint propagation for provenance generation, though at high overhead. PROGRAPHER [13] proposes GPU-accelerated graph construction for high-throughput provenance pipelines. Collberg et al. [72] extend document provenance using edit history tracking. These systems span various platforms and layers, yet they often require intrusive instrumentation, lack support for distributed coordination, or fail to adapt to multi-tenant, containerized settings.

(iii) Causality and Systems

Causality has been studied as a foundational concept in event reasoning and fault attribution. CauseInfer [73] models causality using Granger inference but is limited to numerical telemetry and lacks compatibility with log-based systems. Falcon [61] traces network context but is unsuitable for detailed file I/O tracking. Horus [59] correlates application logs with event traces using causal models but does not support container abstraction. CAT [74]

emphasizes explainability in causal modeling, while HolisticRCA [75] proposes end-to-end causal graphs for observability but lacks real-time integration. These causality systems advance the state of log interpretation and fault tracing but typically rely on temporal ordering or heuristics rather than kernel-level causality, limiting their accuracy in high-concurrency environments.

(iv) Observability in Distributed Systems

Modern observability platforms such as Datadog [35], Dynatrace [36], Aqua [76], and New Relics [77] provide robust infrastructure monitoring, log aggregation, and distributed tracing across containerized environments. These tools offer real-time dashboards, anomaly detection, and service-level visibility, integrating seamlessly with orchestrators like Kubernetes. However, they primarily focus on operational metrics and lack access to low-level system events such as system calls, file I/O, and inter-process communication that are essential for detailed forensic analysis or provenance reconstruction. Academic research like Stitch [78] and LProf [68] attempt to reconstruct execution flows by analyzing structured application logs or combining static binary analysis with runtime profiling. While they offer partial causality reconstruction, Stitch requires application instrumentation, and LProf is limited to controlled binaries, reducing their applicability in dynamic microservice environments. A common limitation across both commercial and academic solutions is their abstraction level. These tools do not capture kernel-level interactions, lack support for causally consistent event ordering, and are unsuitable for fine-grained security monitoring. Additionally, commercial platforms are proprietary, limiting extensibility and integration with custom analysis frameworks. In summary, while existing observability solutions are effective for performance and operational monitoring, they are inadequate for security-focused observability in distributed systems. Their inability to capture semantic, causal, and system-level context highlights the need for open, extensible frameworks that combine application- and kernel-level logs to support provenance-based attack detection and root cause analysis.

Table 2.3 Comparison of Observability Platforms

Reference	Open Source	Container Visibility	Kernel-level Observability	Custom Provenance Support
Datadog [35]	✗	✔	✗	✗
Dynatrace [36]	✗	✔	✗	✗
New Relics [77]	✗	✔	✗	✗
Aqua [76]	✗	✔	✗	✗
Stitch [78]	✔	✔	✗	✗
LProf [68]	✔	✔	✗	✗

(v) Use of eBPF in Logging

Extended Berkeley Packet Filter (eBPF) has transformed observability by enabling safe, high-performance, in-kernel programmability without modifying kernel code. It allows developers to attach programs to system events (e.g., syscalls, tracepoints, kprobes), making it possible to capture telemetry with minimal overhead. Tools like Tracee [40] and Tetragon [79] leverage eBPF for runtime syscall tracing, container monitoring, and security policy enforcement, providing valuable insights into process behavior and attack detection. However, they are primarily focused on security auditing and do not capture application-level context or causally order events across hosts. Sysdig [64] uses eBPF and kernel probes to collect telemetry from running systems but faces challenges in isolating container-level events and lacks support for multi-layer log correlation. eAudit [63] focuses on syscall-level auditing using eBPF in monolithic applications, providing low-overhead visibility but limited adaptability to microservices. Other works extend eBPF's use for distributed logging and optimization. XRP [39] and FuzzyLog [80] explore log consistency and distributed coordination using kernel-level hooks, while HardLog [81] focuses on secure system audit trails. [82] and [38] use eBPF to trace container network communications. Tools like Electrode [83] and TPC [84] apply eBPF to optimize distributed protocols and transport-layer behavior. Despite their effectiveness, these tools generally lack multi-layer log fusion, inter-host causal ordering, and semantic encoding necessary for constructing provenance graphs. There remains a gap in leveraging eBPF for unified, context-rich observability in microservice environments.

Table 2.4 Comparison of eBPF-based Logging Tools

Reference	Year	eBPF-based	Multi-layer Log Support	Real-time Processing	Microservice-aware
Tracee [40]	2021	✔	✗	✔	✗
Tetragon [79]	2022	✔	✗	✔	✗
Sysdig [64]	2016	✔	✗	✔	✗
eAudit [63]	2023	✔	✗	✗	✗
RISE [82]	2020	✔	✗	✗	✔
Protocol-aware Logging [38]	2020	✔	✗	✗	✔
Electrode [83]	2023	✔	✗	✔	✔
XRP [39]	2022	✔	✗	✗	✗

Limitations and Scope

Most existing observability systems either rely on commercial software, lack inter-host correlation, or provide limited support for real-time, multi-layer, and sandbox-compatible logging. Kernel-based frameworks like Tracee and eAudit are single-host and do not offer semantic fusion of logs. Furthermore, static and proxy-based methods cannot support runtime adaptability or pluggability required in microservice orchestration platforms.

2.2.3 Provenance-based Threat Detection and APT Analysis

The use of provenance for detecting advanced cyber threats has emerged as a critical strategy in recent years, particularly in microservice-based architectures. Provenance graphs, which model system behavior through causal relationships between entities such as processes, files, and sockets, offer a comprehensive foundation for analyzing complex attacks such as Advanced Persistent Threats (APTs). In this section, we survey the landscape of provenance-based security, organized into three key directions: (i) provenance-guided threat analysis, (ii) machine learning-based APT detection, and (iii) the unique challenges of threat analysis in sandboxed and containerized microservice environments.

(i) Provenance-Guided Attack Analysis

Provenance graphs have proven highly effective in reconstructing and explaining multi-stage attack scenarios. ATLAS [41] represents one of the early systems that uses NLP techniques to interpret heterogeneous logs and produce structured attack stories, reduc-

ing analyst burden and improving forensic timelines. It applies causality inference over both application and system-level logs to provide a coherent and interpretable view of potential attack campaigns. OmegaLog [9] enhances this approach by building a universal provenance graph that bridges application traces with system events while maintaining causal and temporal integrity. By leveraging control flow integrity, OmegaLog ensures that graph edges reflect meaningful information flow, which enables both real-time detection and post-hoc reconstruction of stealthy APTs. AIQL [42] proposes a powerful query language specifically designed for forensic search over provenance graphs. This domain-specific language allows security analysts to encode behavioral patterns associated with known APTs and efficiently search for matching subgraphs, significantly accelerating the process of root cause analysis and threat hunting. To address the issue of overwhelming dependency graphs during attack investigations, BackPropagation [85] introduces a graph reduction mechanism based on weighted impact propagation. This technique helps filter out less relevant edges, focusing attention on the most critical paths involved in the attack's propagation, thereby improving both performance and clarity during investigation. Together, these works demonstrate that provenance-guided approaches can yield highly interpretable, semantically rich, and causally complete models of system behavior, though most are optimized for host-level or monolithic systems rather than containerized microservices.

(ii) Machine Learning-based APT Detection

With the increasing sophistication of APTs, machine learning techniques have been applied to provenance graphs to automate the detection of malicious activity. Deepro [43] uses graph neural networks (GNNs) to learn deep representations of provenance graphs and classify nodes as benign or malicious. The model captures complex relationships in provenance data, allowing it to detect nuanced patterns that may not be apparent through rule-based techniques. AttRSeq [44] takes a sequential learning approach by modeling event traces as attention-enhanced LSTMs. By focusing on causal sequences of events, the system can effectively detect long-range dependencies and subtle anomalies characteristic of APT behavior. Its attention mechanism allows it to weigh more critical events and suppress irrelevant noise in execution traces. ANUBIS [46] presents a Bayesian neural network

framework for anomaly detection that not only provides predictions but also quantifies uncertainty. This is particularly useful in microservice environments where data can be noisy or incomplete. ANUBIS uses provenance traces from distributed services and learns probabilistic behavior patterns to identify attacks with a measure of confidence, supporting both interpretability and robustness. Liu et al. [45] combine causal inference with deep learning by modeling evolving graph structures that represent interactions among entities. Their approach allows for predictive reasoning about how threats may progress through a system, providing a forward-looking defense strategy. By learning both the structural and temporal evolution of attacks, their model is capable of anticipating malicious activity before full execution. Liang et al. [86] extend this line of work to distributed cyber-physical systems by introducing an event-triggered communication model for anomaly detection. Their decentralized approach enables collaborative monitoring across multiple agents, even in adversarial environments where some components may be compromised. It supports resilient fault detection without relying on centralized control, making it applicable to real-world, large-scale microservice architectures.

Collectively, these learning-based systems demonstrate the power of graph modeling and sequential learning in identifying persistent and evasive attacks. However, many of them are evaluated in limited experimental settings and often assume clean, labeled datasets, which may not be practical in dynamic production environments.

(iii) Challenges with Application Sandboxing

While provenance-based analysis has shown great promise, applying these techniques in sandboxed or containerized environments remains a formidable challenge. Traditional audit tools like Linux Audit [87] offer limited visibility into containerized applications due to namespace isolation and restricted access to kernel-level events. This blind spot can allow attackers to operate within containers without being observed by host-level logging tools. SCADS [88] attempts to monitor system call traces in Linux environments but is primarily tailored for monolithic applications. It fails to capture inter-container interactions or the semantics of distributed workflows, limiting its effectiveness in microservice deployments. POIROT [57] and HOLMES [20] build attack graphs in near real-time by analyzing audit logs and applying behavior-based detection strategies. While they excel in kernel-

space causality tracking, their models are tightly coupled to single-host execution contexts and cannot generalize to orchestrated container environments. UNICORN [12] and ORCHID [89] focus on enhancing analyst usability by producing summarized provenance graphs that retain essential behavior patterns. However, they lack support for ephemeral containers and cannot integrate cross-host context, which is crucial for capturing the lateral movement of APTs across services. Zhu et al. [90] attempt to bridge this gap with KubProv, a Kubernetes-aware provenance system that captures pod-level event sequences and maps them to service graphs. Although this work brings orchestration context into attack analysis, it relies heavily on Kubernetes APIs and does not generalize well to other container platforms like Docker Swarm or serverless frameworks.

Furthermore, most existing provenance systems require root access or kernel module insertion, which introduces operational complexity and security risks in production environments. In multi-tenant or shared infrastructure scenarios, such invasive instrumentation is often infeasible. These constraints highlight the urgent need for a lightweight, pluggable, and orchestration-aware provenance system that can capture causally complete traces across microservices without modifying the host or application code.

Limitations and Scope

From the above review, several clear limitations emerge. First, many provenance-based detection systems operate on static assumptions about the execution environment, failing to adapt to ephemeral and sandboxed microservices. Second, few systems support the causal reconstruction of attacks that span multiple services or hosts—a key requirement for analyzing real-world APTs. Third, kernel-level tools often lack visibility into containers, while user-space logging is too coarse-grained to support detailed forensic analysis. Fourth, while learning-based methods provide strong detection performance, they often require labeled training data, which is scarce in security-sensitive domains. Lastly, few existing tools offer end-to-end solutions that combine fine-grained observability with scalable, real-time anomaly detection across containerized microservice platforms.

These limitations collectively motivate the design of *μProv*, a provenance-driven threat detection framework capable of operating across sandboxed microservices by capturing

fine-grained, causally ordered execution traces. Our system is designed to address the key gaps in container compatibility, semantic modeling, and scalable graph-based analysis, making it suitable for modern microservice-based infrastructures.

2.2.4 Graph-based Learning over Provenance for Anomaly Detection

As the complexity of microservice systems increases, so too does the sophistication of cyber-attacks, particularly Advanced Persistent Threats (APTs), which evolve across multiple layers and hosts. Provenance graphs, which encode causal relationships among system entities, have emerged as a rich and expressive abstraction for modeling such behaviors. Recent research has turned to graph-based learning, particularly Graph Neural Networks (GNNs), to automate the detection of anomalous activity within these provenance structures. In this section, we classify existing literature into four themes: (i) provenance-based attack detection infrastructure, (ii) graph neural networks for anomaly detection, (iii) APT modeling through graph-based learning, and (iv) the specific challenges of modeling provenance in microservice environments.

Table 2.5 Comparison of Anomaly Detection Systems

Reference	Semantic Encoding	Temporal Encoding	Detection Granularity	Supervision
UNICORN [12]	✗	✔	Graph	Unsupervised
ProvDetector [51]	✗	✗	Graph	Unsupervised
Deepro [43]	✔	✗	Node	Supervised
Prov2Vec [49]	✗	✗	Graph	Unsupervised
ThreatRACE [50]	✔	✔	Node	Supervised
AttRSeq [44]	✔	✔	Sequence	Supervised
ANUBIS [46]	✔	✗	Microservice	Supervised

(i) Provenance-based Attack Detection Infrastructure

Several works have focused on constructing provenance graphs as the foundation for anomaly detection. OmegaLog [9] introduces a unified provenance representation by integrating system call traces and application-level logs to capture semantic and causal relationships within monolithic systems. It enhances the expressiveness of provenance graphs by incorporating control flow and multi-layered context, enabling accurate reasoning over sus-

picious event chains. UNICORN [12] applies Random Forest classifiers on handcrafted graph features to detect APTs by capturing deviations in graph structure indicative of attack behaviors. This method emphasizes edge labeling and subgraph-based reasoning, though it is primarily designed for homogeneous log environments. Deepro [43] presents a significant advancement by applying GNNs to provenance graphs, learning deep representations of node interactions, and enabling node-level classification of benign versus malicious activities. The system is capable of capturing nuanced dependencies between processes and resources, and is particularly effective in modeling stealthy APT patterns. ATLAS [41] takes a complementary approach by using natural language processing to correlate heterogeneous logs and reconstruct attack narratives. It applies machine learning to infer causal relations and reduce analyst fatigue by summarizing large-scale telemetry data into concise attack stories. MARLIN [91] proposes an online approach to provenance tracking by propagating tags across streaming graphs, allowing for real-time detection of anomalous flows. However, it lacks support for tracking provenance across sandboxed containers and microservices. Collectively, these systems highlight the foundational role of provenance graphs in capturing system behavior and facilitating automated analysis. Yet, most are limited to host-level abstraction and fail to account for the distributed, dynamic nature of microservice-based deployments.

(ii) GNNs for Anomaly Detection

Graph neural networks have shown significant promise in detecting anomalies by learning structural and relational features from provenance graphs. Prov2Vec [49] proposes an embedding-based model that transforms provenance graphs into fixed-length vectors using statistical features, allowing unsupervised clustering of benign and malicious traces. However, it lacks semantic node typing and fails to encode causal or temporal relationships directly. ThreatRACE [50] addresses this limitation by introducing temporal attention into the learning pipeline, allowing the model to focus on time-sensitive interactions that often characterize stealthy threats. ProvDetector [51] exposes the vulnerability of existing GNN-based systems to mimicry attacks and proposes the incorporation of semantic encoding to distinguish between syntactically similar but semantically distinct activities. This work emphasizes the importance of modeling richer context and behavior rather than relying solely

on structural features. NoDoze [92] introduces a summarization mechanism to reduce the scale of provenance graphs into "alert trees," enabling scalable triaging by human analysts without losing key causal relationships. ORCHID [89] builds on this idea by generating interpretable graph summaries through hierarchical clustering and node relabeling, improving analyst usability without compromising detection accuracy. These GNN-based models demonstrate the ability to learn from provenance data in both supervised and unsupervised settings. However, most of them are built on homogeneous graph representations, lack semantic richness, and do not scale well to large multi-tenant systems where provenance graphs are both dynamic and high-volume.

(iii) APT Modeling and Inference using Graphs

Beyond anomaly detection, several systems have been developed to model APT campaigns using graph-based inference techniques. AIQL [42] introduces a domain-specific query language for expressing and detecting APT behaviors over provenance graphs. This enables analysts to encode known attack signatures as logical patterns and search for matching subgraphs in execution traces. BackPropagation [85] proposes a graph pruning technique that uses weighted propagation to eliminate low-impact edges, helping analysts focus on causally significant paths during post-attack investigation. AttRSeq [44] models sequences of events as attention-based LSTMs trained on causally ordered logs, allowing detection of APT campaigns that unfold over long periods and across multiple components. ANUBIS [46] applies Bayesian deep learning to capture both prediction and uncertainty in microservice security monitoring, making it suitable for environments where ground-truth labels are noisy or incomplete. Liang et al. [86] propose a decentralized model for cyber-physical systems that uses event-triggered communication graphs to diagnose attacks in real-time, even in adversarial settings where some nodes may be compromised. These systems offer robust modeling capabilities for complex, multi-stage attacks. However, they often assume access to well-structured logs or labeled datasets, and may not generalize well to the ephemeral, distributed execution environment characteristic of microservices.

(iv) Challenges in Microservice Provenance

Despite promising results in graph-based learning, several limitations remain in modeling provenance effectively within microservice ecosystems. Traditional systems like SPADE [52], Practical Provenance [47], and Trustworthy Provenance [48] offer robust mechanisms for provenance capture and storage but are heavily dependent on host-level kernel instrumentation. These systems do not scale to containerized workloads where isolation and orchestration complicate system call visibility. Tools like POIROT [57] and HOLMES [20] attempt real-time correlation of events for detecting ongoing attacks, but their graph construction mechanisms struggle with the high churn of containerized processes and networked services. UNICORN [12] and ORCHID [89], while effective in summarizing graphs for analyst usability, are designed primarily for per-host provenance and do not encode service-level semantics or interactions. Zhu et al. [90] introduce Kubernetes-aware provenance modeling to capture container lifecycle events and inter-service dependencies. However, this system depends on specific orchestration platform APIs, limiting its portability across diverse cloud environments. In summary, existing provenance models often assume a static execution context, lack support for dynamic service discovery, and cannot capture causally consistent event sequences across distributed hosts. These constraints significantly hinder their applicability for real-time attack detection in microservice-based architectures.

Limitations and Scope

The works surveyed in this section reveal a strong foundation for applying graph-based learning to provenance data for security analytics. However, key limitations remain. Most systems are host-centric and do not scale to distributed or containerized microservice environments. Graph representations often lack semantic heterogeneity and temporal encoding, both crucial for modeling complex attack behaviors like APTs. Existing GNN-based models typically rely on supervised learning, which needs curated datasets rarely available in security-sensitive settings. Many solutions also require specific configurations or instrumentation unsuitable for production. These challenges motivate *μProvGAE*, which builds enriched, causally consistent provenance graphs and uses unsupervised graph learning to detect anomalies in multi-host microservice environments.

2.3 Summary

This chapter has presented a comprehensive review of existing research on provenance-based logging, observability, causality analysis, and attack detection in both monolithic and microservice architectures. We examined various categories of related work, starting with system logging and provenance graph construction, where many existing approaches rely on static or kernel-instrumented modules and are not suited for dynamic, container-based environments. We then discussed causality and observability systems, noting that while some frameworks attempt to capture causal relationships, they lack support for inter-host coordination and full-stack log integration. The use of eBPF has enabled new directions in observability, but existing eBPF-based tools are either focused on specific domains like networking or auditing, or they fail to integrate application-level context with kernel-level events. In the domain of attack detection, provenance graphs have proven useful for modeling system behaviors, yet most solutions focus on individual services or hosts, overlooking the interconnected nature of attacks in microservices. We also explored GNN-based anomaly detection methods, highlighting their limited support for semantic heterogeneity, temporal modeling, and microservice-level granularity. Across all these categories, we observe a common set of limitations, including host-centric assumptions, lack of support for sandboxed environments, reliance on supervised learning, and poor scalability in distributed settings. These gaps motivate the need for a unified observability and attack detection framework that can construct semantically rich, causally ordered provenance graphs and apply unsupervised learning to detect anomalous behaviors across containerized microservices. This serves as the foundation for our proposed solution, which is presented in the following chapters.

Chapter 3

Distributed Provenance Tracking over Serverless Applications

Modern service-oriented architecture adopts DevOps [93,94] practices and technologies to provide Software as a service (SaaS) [95] by leveraging distributed cloud infrastructure. Service deployment on top of the cloud widely adopts serverless computing (SLC) [96] to reduce operational expenditure whenever the service computations are stateless, elastic, and possibly distributed. Micro-services deployed on top of SLC architecture provide an abstraction of the underlying infrastructure where the developer can write, deploy, and execute the code without configuring and managing the shared environment [96–99]. However, the available serverless-specific industry solutions [100–102] provide limited support for error reporting, execution tracing, and provenance tracking. Consequently, developers can only provide little attention to log vital forensic information. Some of the third-party observability tools [28–31] support distributed tracing as well as cost analysis features by instrumenting the source code. However, these tools only support applications developed using a particular programming language. So, it is difficult to analyze the actual behavior of these microservices.

Provenance Graphs: The non-invasive [1] frameworks rely on the logs generated by

[1]The non-invasive tools neither inject any piece of code inside the micro-service/container instances nor inject any special services into the SLC platform.

applications to identify the execution states. Since most of the production-grade microservices are chosen from an available stable release, the executable files already contain meaningful log messages that can be used to identify event handling loops. In the domain of system security, provenance data is the metadata of a process that records the details of the origin and the history of modification or transformation that happened over time throughout its lifecycle [9, 25, 46, 103–105]. The graph generated from this information is called the *provenance graph* of a process which is a causal graph that stores the dependencies between system subjects (e.g., processes) and system objects (e.g., files, network sockets). For example, an application event may generate a separate application log, error log, and operating system calls specific log, etc. In this context, a provenance graph is a Directed Acyclic Graph (DAG) where individual log entries are the nodes, and the edges represent the causality relationship between the log entries. During an attack investigation, an administrator queries this graph to find out the root cause and ramifications of an event. While a malicious entity performs some illegal events, the corresponding system logs are recorded as the provenance data. For example, when a compromised process tries to open a sensitive file, the OS level provenance can record the file-open activity, which can be referred for vulnerability analysis whenever an attack is detected in the system [48, 106]. Real-world enterprises widely use kernel or OS level information (logs) to perform provenance analysis to monitor their systems and identify the malicious events performed back in time.

3.1 Limitations of Existing Works and the Research Challenges

There have been several works that attempted to generate the provenance graphs by combining system and application logs together [9, 25, 46, 104, 105, 107]. However, these works primarily considered a monolithic application running directly over the Kernel, and thus combining the system log with the application log is not difficult. In contrast, the individual log files for each microservice over an SLC application are physically distributed across the entire ecosystem, which makes the generation of the provenance graph non-trivial. In such a scenario, a Universal Provenance Graph (UPG) that combines the interactions among all the microservices can provide a meaningful platform for distributed provenance

tracking. A few existing works [25, 108] have tried to address the issue of encoding all forensically-relevant causal dependencies regardless of their origin. Nevertheless, we have some non-trivial challenges in constructing a UPG for an SLC application.

• Challenge (1) – *Combining application logs from different micro-services:* Different micro-services generate separate logs that vary across the format for log messages, naming of the events and process descriptors, timestamp formatting, etc. Combining these logs towards generating the UPG is non-trivial as they may result in confounding nodes and edges in the graph.

• Challenge (2) – *Combining the system log with the application logs:* The next challenge comes in terms of combining the application logs with the system log (kernel `audit log`). The first issue is that the container-based sandboxing uses a common process identifier (`pid`) space; thus, mapping the kernel process logs with the micro-services event logs is not straightforward, particularly when a microservice runs a multi-threaded process.

• Challenge (3) – *Identification of execution units:* As different micro-services may come from different production endpoints, there exists heterogeneity in terms of their implementation standards. Thus it is non-trivial to identify the functions or execution units that generate a specific entry in the log. In the same context, it is also difficult to identify the event handling loops, as the loops might produce asynchronous log entries. This problem magnifies in the case of SLC as the microservices are deployed in different sandboxed environments.

Owing to the above challenges, in this chapter, we develop a provenance tracing system, *DisProTrack* that generates a UPG which helps in root cause analysis of an event running on serverless by combining the system provenance with the application's container logs. The core idea behind *DisProTrack* is to judicially use the applications' CFG to avoid the runtime log tracking complexities.

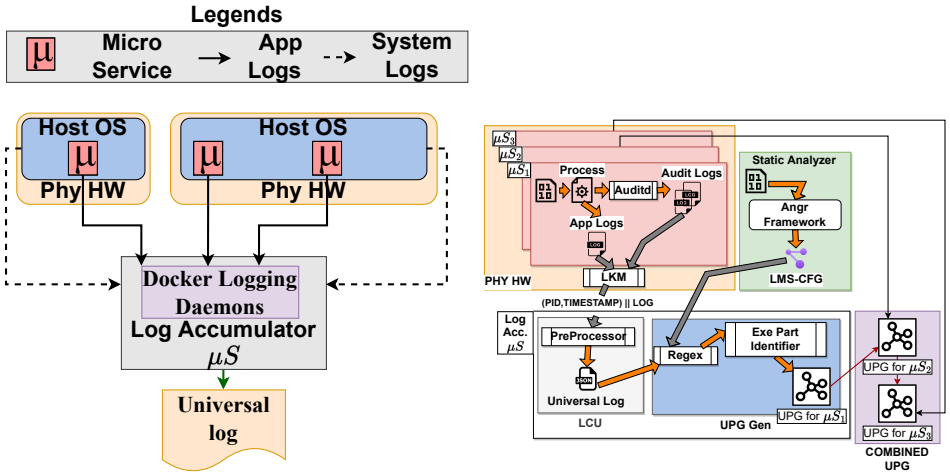
3.2 Our Contributions

In contrast to the existing works, our contributions in this chapter are as follows.

• Contribution (1): *Design of the Universal Provenance Graph (UPG) from application and system logs.* – We implement a static analyzer module that generates the application-specific *Log Message String - Control Flow Graph* (LMS-CFG) from the application binaries. The LMS-CFG provides a profile of the application. We provide a novel approach for constructing a UPG from application logs and system logs using the LMS-CFG profiles for different application micro-services.
• Contribution (2): *Runtime execution unit identification.* – We develop a Linux Loadable kernel module (LKM) that can intercept the system calls generated during execution time to identify the semantic relationship between the system logs and the application logs. Furthermore, we propose a heuristic to segregate execution units that are challenging in a distributed system. Our proposed heuristic identifies the system calls (syscalls) to mark the application's event handling loops by tracing back the application binaries. These event handling loops can be refereed during the runtime partitioning of execution units across the micro-services.
• Contribution (3): *Utilization of Regular Expression to improve search efficacy.* – Instead of storing the raw log messages in the UPG, we propose conversion and storage of an equivalent regular expression. This method improves the matching accuracy of log messages during the investigation phase and reduces the runtime search complexity by providing a faster response time. This method also reduces dependency explosion by decreasing the number of nodes in the generated UPG.
• Contribution (4): *Implementation and evaluation.* – We have implemented the proposed framework, which can be deployed as a microservice on top of the SLC without instrumenting the source code of the applications. We have made the implementation open-sourced[2]. Based on the experimental evaluation of *DisProTrack* with several benchmarked SLC applications, we found that the proposed method works well for identifying adversary activities. The framework has a minimal memory footprint (in the order of KB) and responds within $20s$-$30s$. The efficiency and efficacy of *DisProTrack* have also been tested with a proof-of-concept SLC application scenario.

[2] https://github.com/usatpath01/DisProTrack (Accessed: May 5, 2026)

3.3 *DisProTrack* Overview



(a) Environment Setup **(b)** *DisProTrack* Execution Flow

Figure 3.1 *DisProTrack* Overview

The proposed *DisProTrack* is an open source provenance tracking system for containerized SLC as shown in fig. 3.1a. *DisProTrack* accumulates the system and application logs of all the micro-service containers (μS) to generate the underlying directed edge-labeled provenance graph. The nodes of the provenance graph represent the system artifacts, for example, processes, socket connections, files, etc. The edges represent the causal dependencies between the nodes. Each edge is labeled with the generated system call (syscall) events (e.g. *read*, *write*, *connect*, *exec*, etc.) as shown in fig. 3.2. The accumulated logs are merged together to create a *universal log* which is further used to identify the nodes of the provenance graph. Additionally, the framework also analyzes the Control flow graph (CFG) of the individual applications to understand the event handling loops before the deployment. So, the provenance graph generation requires two phases and, depending on that, *DisProTrack* is sub-divided into two major components; (a) *static analyzer* and (b) *runtime engine* as shown in fig. 3.1b.

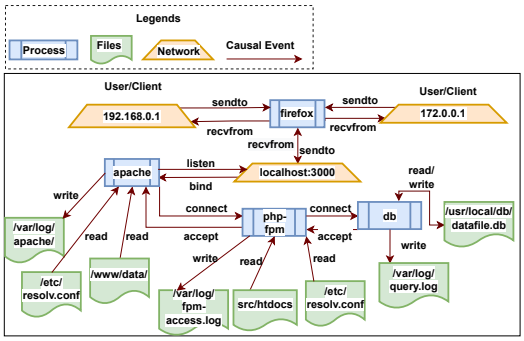


Figure 3.2 An Example of System Provenance

3.3.1 *DisProTrack* Static Analyzer

The static analyzer module analyzes the application executables and generates a semantic relationship between multiple *Log Message String*(LMS). Here, we define an LMS as a string present in the executable responsible for printing some log message. Typical LMS contains format specifiers, error codes, debug level identifiers, etc. Since we only require the causal relationships between LMS, the static analyzer identifies only the *Log Message Generating Functions*(LMGF).

Definition 1 (Log Message Generating Functions)**.** We define a *Log Message Generating Function* (LMGF) as a library function that is directly used for printing an LMS in either terminals, specific log store files, or log databases.

For example, consider the following C code snippet with a popular logging library `Log4C`.

```
log4c_category_log(NULL, LOG4C_PRIORITY_ERROR,"Hello World!");
```

Here the LMS is `Hello World!` and the LMGF is `log4c_category_log`. More details about LMGF is described in section 3.4.1.

Definition 2 (Log Message String CFG)**.** A LMS-CFG is formally defined as a directed graph $G' = (V', E')$ where $\underset{e'_{i,j} \in E'}{\forall} e'_{i,j} = (v'_i, v'_j) : v'_i, v'_j \in V'$ represents a directed edge between v'_i, v'_j where each $v' \in V'$ represents an LMGF. G' is constructed from CFG $G = (V, E)$ such that, $V' \subseteq V$ and, $\underset{e' \in E'}{\exists} e' = (v'_i, v'_k) : v'_i, v'_j, v'_k \in V', \nexists_{v'_j} v'_j \in \mathfrak{P}(G, v'_i, v'_k)$. In this case $\mathfrak{P}(G, v'_i, v'_k)$ represents the directed path from v'_i to v'_k in G.

An example of a CFG and the corresponding LMS-CFG is provided in fig. 3.3, where the constructed LMS-CFG contains only the LMS nodes from the CFG.

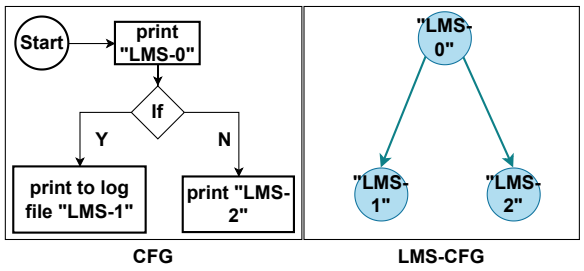


Figure 3.3 An Example of System Provenance

Contextualization of application log and system log (handling Challenges 1 & 2)

The LMS-CFG is stored separately and frequently consulted by the runtime engine. During the execution, when the different levels of logs are generated, the constructed LMS-CFG is used to understand the relationship among those log messages. Based on the relationship, the logs coming from different sources are combined together (details in section 3.4.2.ii).

3.3.2 *DisProTrack* Execution Path (Runtime Analyzer)

During the execution of the system, the applications generate multiple logs depending on the user's activities. So the task of the runtime engine is to collect and contextualize the log messages generated at the systems and application levels. We assume that the micro-services containers have `auditd` [3] installed for monitoring system-level logs. This assumption does not violate the *"without instrumentation of the application source code"* constraint, as it is straightforward to add an `auditd` layer to the existing containers without knowing anything about the application source codes.

Tracing the execution units from processes belonging to different micro-services (handling Challenge 3)

To avoid confusion during the contextualization process of the accumulated logs, we append the Process ID (PID) of the process responsible for generating the log and the timestamp to add the semantic context to individual log entries. For this purpose, we develop

a Loadable Kernel Module (LKM) which intercepts all the write syscalls caused by log printing functions to extract the PID information and timestamp of the system and append it to the log preamble. This LKM is deployed in the bare metal infrastructure where the containers are executing (details in Section 3.4.2.ii).

Dependency explosion and handling confounding root causes (handling Challenge 4)

During the execution of the applications, the runtime engine periodically fetches the marked log entries from the micro-services. It generates the Universal Provenance Graph (UPG) after consulting the LMS-CFG. The LMS-CFG provides a relationship between the applications and file, which is exploited to construct UPG as depicted in fig. 3.2. The details of the UPG construction procedure are described in the next section. The generated UPG is consulted by the system administrators for the system provenance tracking. The root cause of any suspicious log entry can be identified by backtracing the UPG (details in line 29).

3.4 Components of *DisProTrack*

In this section, we describe the design of individual components of *DisProTrack*. As mentioned previously, *DisProTrack* is subdivided into two parts; (a) *Static Analyzer*, and (b) *Runtime Engine*.

3.4.1 Static Analyzer

The static analyzer takes individual micro-service's application binaries as input and constructs the corresponding LMS-CFG for that micro-service application. A typical application will contain a set of statements to print the LMS with syscalls in between. In our proposed framework, we load the executable application binary and use the python `Angr` module [109, 110] to identify the CFG from it. The generated CFG is a directed graph having the basic instruction blocks (syscalls, printing of LMS, etc.) as nodes, and the edges of the graph represent jump/call/return statements from one block to another. Let

the CFG be represented as $G = (V, E)$, where V and E are the nodes and the edges of the CFG, respectively. We then use the same `Angr` module to extract all the nodes $\mathbb{F}$ from the CFG $G(V, E)$ corresponding to various function calls. One significant issue with the generated CFG is that it does not always provide the complete graph due to the missing hardware-dependent features and system call information. Therefore, in this case, we only concentrate on the LMGF. Typically the stable versions of the applications use standard LMGF (e.g. `printf`, `log4c`, `syslog`, etc.). Let $\mathcal{L}$ be a list of standard LMGF names. We find $\mathbb{L} \subseteq \mathbb{F}$ where $\mathbb{L}$ contains the LMGF from $\mathcal{L}$. We construct a LMS-CFG $G'(V', E')$ from $G(V, E)$ with the help of $\mathbb{L}$.

We propose algorithm 1 to convert $G = (V, E)$ to $G' = (V', E')$ for a given $\mathbb{L}$ as the input. Each node in G represents a sequence of instructions. If a node v contains an LMGF name, then the algorithm extracts the arguments of the LMGF, which has been defined as an LMS apriori. For example, for a given LMGF `fprintf`, consider the following log entry function.
"`fprintf(stderr, "AH00526: Syntax error on line %d of %s:");`"
In this case, the algorithm extracts the LMS as `"AH00526: Syntax error on line %d of %s:"`, which is the constant string reference passed as an argument to the LMGF.

During the construction of LMS-CFG, we need to identify the caller functions of the LMGF, which is a time-consuming operation. Therefore, we limit the depth of backward tracing up to a certain threshold (BT), as shown in Algorithm 1:Line 13. The optimal value of BT depends on the complexity of the generated CFG, which we shall discuss during the experimental evaluation (section 3.5.2). For accurate identification of the LMS, we need to avoid the programming language-specific format specifiers. For that, we replace the format specifier of LMS with equivalent regular expressions (see Algorithm 1:Line 10). For example, consider the LMS as shown earlier: "`AH00526: Syntax error on line %d of %s:`". The regular expression equivalent to this LMS becomes "`AH00526: Syntax error on line -?[0-9]+ of .*:`", where `%d` and `%s` are replaced with `[0-9]+` and `.*`, respectively. Additionally, we mark the starting and ending LMS positions of the event handling loops with a flag. This generated and marked LMS-CFG is used by *Runtime Engine*, explained next.

Algorithm 1: CFG to LMS-CFG Generation

Procedure `Main`
  Input: $G(V, E)$: CFG, $\mathbb{L}$: Set of Log message Generating Functions (LMGFs) in $G(V, E)$
  Output: LMS-CFG $G' = (V', E')$
  Initialization:
    $V' \leftarrow \varnothing, E' \leftarrow \varnothing$
  for $v \in \mathbb{L}$ **do**
    /* The Angr tools return whether a node is a loop */
    if *v has a loop* **then**
      /* For a loop, we need to find out all the Log message string (LMSes) ($\mathbb{A}$) that are printed through the loop. */
      $\mathbb{A} \leftarrow$ `BackTraceOptimization`$(BT_{th}, G(V, E), v)$;
      /* Construct the subgraph $G_A(V_A, E_A)$ for A */
      $G_A(V_A, E_A) \leftarrow$ `CreateSubGraph`(A_i);
      $V' \leftarrow V' \cup V_A$;
      $E' \leftarrow E' \cup E_A$;
  for $v' \in V'$ **do**
    Identify format specifiers in v' and replace them with suitable Regular Expressions;
  return $G'(V', E')$;

Function `BackTraceOptimization`
  Input: BT: Backtrace Threshold, $G(V, E)$: CFG, v: A node from $\mathbb{L}$ having a loop
  Output: $\mathbb{A}$: A set of LMSes in the loop
  Initialization:
    $\mathbb{A} \leftarrow \varnothing$
  if *v has a set of LMSes $\{l_0, \cdots l_k\}$ printed through the loop with syscalls in between the LMSes* **then**
    /* We consider the loops having a syscall and associated LMSes */
    $\mathbb{A} \leftarrow \mathbb{A} \cup \{l_0, \cdots, l_k\}$;
    return $\mathbb{A}$;
  else
    /* The loop within v does not print any LMSes */
    do
      /* Backtrack to the LMGF that called v, until the backtrack threshold BT is reached. */
      if *$\langle \bar{v} \rightarrow v \rangle$ is a directed edge in $G(V, E)$* **then**
        if *$\bar{v}$ has a loop with a syscall and a set of LMSes $\{l_0, \cdots l_k\}$* **then**
          $\mathbb{A} \leftarrow \mathbb{A} \cup \{l_0, \cdots, l_k\}$;
          return $\mathbb{A}$;
      $v \leftarrow \bar{v}$;
      $BT \leftarrow BT - 1$;
    while $BT > 0$;
  return $\mathbb{A}$;

Function `CreateSubGraph`
  Input: $\mathbb{A}$
  Output: $G_A(V_A, E_A)$: A Subgraph generated from $\mathbb{A}$
  Initialization:
    $V_A \leftarrow \varnothing, E_A \leftarrow \varnothing$
  foreach $\{l_i, l_{i+1}\} \in \mathbb{A}$ **do**
    $V_A \leftarrow V_A \cup \{l_i, l_{i+1}\}$;
    $E_A \leftarrow E_A \cup \{\langle l_i \rightarrow l_{i+1} \rangle\}$;
  return $G_A(V_A, E_A)$;

3.4.2 Runtime Engine

We design *DisProTrack* Runtime Engine as a micro-service that can be deployed in the serverless platform. Since most of the serverless functions are deployed using multiple containers, log collection and analysis are challenging. *DisProTrack* is concerned about two types of logs; (i) Logs generated by the applications (i.e., inside the container) and (ii) System and/or serverless daemon logs (i.e., logs from the physical servers systems) – we call them Syslogs. The major challenge of obtaining a container's internal logs is that as the process spaces of the containers and the host system are isolated, identification of processes and syscalls become difficult. To avoid such issues, we use an audit daemon in each container by deploying a separate image layer[3] over the containers. The audit daemon is used to track the syscalls generated by the applications inside the containers.

However, audit logs provide the container internal PID, which might conflict with the audit logs obtained from different containers. The conflict must be resolved before the aggregation of the system log and application log. Therefore, we develop a LKM which intercepts LMS before it is printed in the log file and appends a unique tag (which is a combination of container ID, PID, and timestamp) to each LMS entry. This unique tag is used to establish a relationship by adding semantic context among the LMS.

The scope of operation for the runtime engine starts whenever the applications start generating logs. We have segregated Runtime Engine into three submodules; (a) Log accumulator, (b) Log processor, and (c) Provenance builder, which are described as follows.

(a) Log Accumulator

Once the log files are generated, the *Log Accumulator* module periodically pulls the log files from all levels and performs operations on them to correlate them with the events. The non-persistent and ephemeral nature of micro-services implies the risk of data loss or the loss of logs generated during the execution phase of the container lifecycle when the container shuts down. Therefore, the proposed module periodically pulls the logs. To

[3] https://docs.docker.com/storage/storagedriver/ [accessed: May 5, 2026]

prevent data loss due to "SIGKILL", we deploy a signal handler inside the deployed image layer to instruct the container to save the logs in a persistent data volume.

(b) Log Processor

Once the logs are accumulated, the *Log Processor* module aggregates the logs collected from various sources. However, simple concatenation of log files does not preserve the semantics relationship. Therefore, we convert the text-based log files into an equivalent JSON format. From the formatted application log files, the PID of the application is extracted from the tag generated by the LKM. Using the PID, the syscalls are identified from the audit logs. Now the LMS and the syscall-generated logs are merged to create an Application-Specific Common Log (ASCL) file such that the application logs appear just before the corresponding Syslogs.

(c) Provenance Builder

At the runtime, the *Provenance Builder* takes the ASCL file and LMS-CFG of a process as input and constructs the corresponding UPG component for that process. The ASCL file contains interleaved application-level logs (⟨ pid, ts, lms ⟩) and Syslog (⟨ pid, ts, syscall, path, exe ⟩) entries. Using algorithm 2, we identify the execution units in the ASCL file with the help of LMS-CFG, where an execution unit represents a sequence of LMSes execution of a process. In this heuristic, we intelligently apply the LMS-CFG to mark the end of an execution unit by using the leaf nodes of the graph. In this heuristic, we assume that the micro-services are *weakly time-synchronized* with a small time drift λ. The bound on λ depends on how frequently the log messages from two different execution units are getting printed. In reality, λ can be in the order of a few hundred milliseconds as printing the logs too frequently is also an overhead for an application.

Extract Execution Units with the help from Log message string - control flow graphs (LMS-CFGs) of application micro-services. Initially, the execution unit is empty. When the algorithm encounters an application-level log entry from the file, which also happens to be the first entry, it performs a regular expression matching on the LMS-CFG to find a node

Algorithm 2: UPG Construction

Procedure `Main`
  Input: $G' = (V', E')$: LMS-CFG, $\mathbb{F}_{pid}$: ASCL file for process pid
  Output: $G^U_{pid} = (V^U_{pid}, E^U_{pid})$: UPG component for process pid
  Initialization:
    $V^U_{pid} \leftarrow \emptyset$, $E^U_{pid} \leftarrow \emptyset$, $\mathbb{U}_{pid} \leftarrow \emptyset$, $\mathbb{E}_{pid} \leftarrow \emptyset$, $\mathbb{G}_{pid} \leftarrow \emptyset$;
    /* $\mathbb{U}_{pid}$: An execution unit denoting the set of LMSes matched with V' */
    /* $\mathbb{E}_{pid}$: Set of system logs corresponding to an execution unit */
    /* $\mathbb{G}_{pid}$: Set of all $\mathbb{E}_{pid}$ from $\mathbb{F}_{pid}$ */
    $end\_unit \leftarrow$ false /* tracks execution units */
  foreach e *in* $\mathbb{F}_{pid}$ **do**
    /* e can be an application-level entry $\langle pid, ts, lms \rangle$ or a syslog entry $\langle pid, ts, syscall, path, exe \rangle$ */
    /* pid: Process ID, ts: Log timestamp */
    /* lms: LMS, $syscall$: Syscall in log */
    /* $path$: System object (dir, socket, etc.) accessed by the executable */
    /* exe: Name of the executable */
    if e *is an application-level entry* **then**
      if e *is the first entry in* $\mathbb{F}$ **then**
        /* Perform a regex match to find a node n from $G'(V', E')$ which matches with e. n denotes the current state of the search. */
        $n \leftarrow$ `FindLMSinGraph`(G', e);
        $\mathbb{U}_{pid} \leftarrow \mathbb{U}_{pid} \cup \{n\}$
      else
        /* Perform a regex match to find a node in the neighbor of n over the graph $G'(V', E')$; the new node becomes the current state of the search */
        $n \leftarrow$ `MatchNeighbourNodes`(G', n, e);
        $\mathbb{U}_{pid} \leftarrow \mathbb{U}_{pid} \cup n$;
      if n *is a leaf node in* $G'(V', E')$ **then**
        $end\_unit \leftarrow$ true /* Seen a block of LMSes logged by the application from a LMGF */
    if end_unit *is true* **then**
      $\mathbb{E}_{pid} \leftarrow \mathbb{E}_{pid} \cup e$;
      $end\_unit \leftarrow$ false /* tracked syslogs for an execution unit */
      $\mathbb{G}_{pid} \leftarrow \mathbb{G}_{pid} \cup \mathbb{E}_{pid}$;
      $\mathbb{E}_{pid} \leftarrow \emptyset$
    else
      /* e is a syslog entry and end_unit is false */
      $\mathbb{E}_{pid} \leftarrow \mathbb{E}_{pid} \cup e$;
  /* Tracked all syslogs for individual execution units, now construct the UPG */
  $partition \leftarrow 0$ /* keep tracks of individual execution units */
  foreach $\mathbb{E}_{pid} \in \mathbb{G}_{pid}$ **do**
    $partition \leftarrow partition + 1$;
    foreach $e \in \mathbb{E}_{pid}$ **do**
      if e *has a valid exe and syscall* **then**
        $V^U_{pid} \leftarrow V^U_{pid} \cup \{e.exe\}$ /* the name of the executable application binary becomes a node */
        $V^U_{pid} \leftarrow V^U_{pid} \cup \{partition \| e.path\}$ /* the $partition$ value appended with the object path accessed in e becomes the second node */
        $E^U_{pid} \leftarrow E^U_{pid} \cup \{\langle e.exe \rightarrow partition \| e.path, e.syscall \rangle\}$ /* an edge is added between the above two nodes with $e.syscall$ as the edge label */

with a match of that entry. If a valid match, say, n, is found, then n becomes the current state. For the rest of the log entries, it performs the regular expression matching with the neighbors of n from the LMS-CFG to find the next state. This step is repeated for all the application-level entries till we find n as a leaf node in the LMS-CFG, which denotes an end unit of the execution unit. The intermediate Syslog entries are added to their respective PID's execution unit E_{pid}. When the end unit becomes true, it implies a block of LMSes has been printed along with a syscall execution. Hence, we save the state of this execution unit in a set G_{pid} and make the execution unit E_{pid} empty for the next set of LMSes to be added. We also set the end unit flag to false.

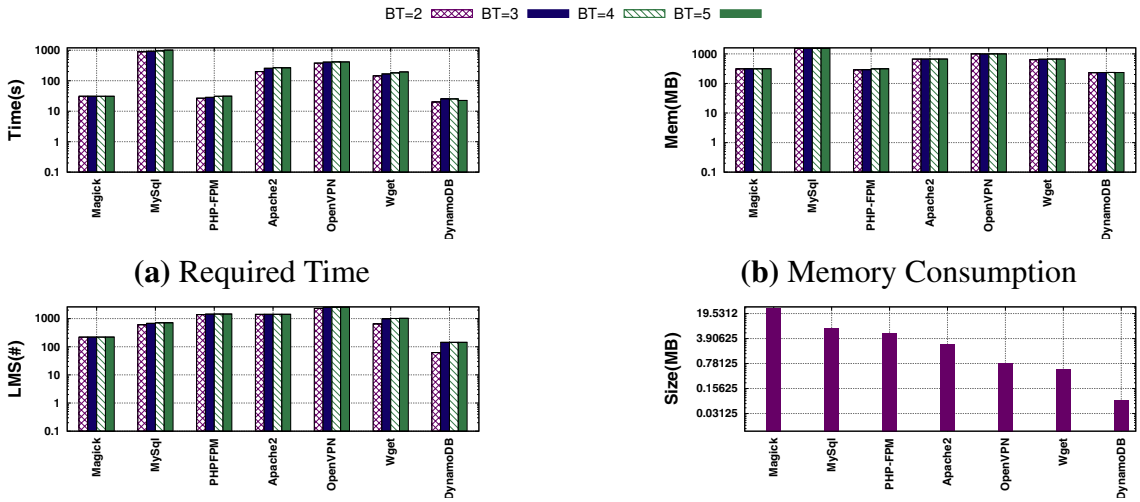
Interconnect execution units. Once all the syslogs for an execution unit is extracted, Algorithm 2 (Line: 23-29) constructs the UPG for that execution unit G_{pid}. We keep track of individual execution units in G_{pid} using a variable called $partition$. For every execution partition in G_{pid}, if an entry e contains exe and $syscall$, then the nodes of the UPG are – (i) the name of the executable application binary ($e.exe$), and (ii) the system objects such as files, socket, directory, etc. ($e.path$) appended with the $partition$ value. The edges are added between the above two nodes, where the corresponding Syscall denotes the edge labels from the Syslog entry ($e.syscall$). The resulting UPG is the union of all the UPG components constructed for each PID.

3.5 Performance Evaluation

The objective of our experimentation is two-fold as follows.

1. Since *DisProTrack* is targeted for serverless applications; resource overhead is a major concern. Therefore, experimentally we want to understand the resource overhead of the framework.

2. We also want to understand how effective *DisProTrack* is for identifying malicious activities.



(a) Required Time **(b)** Memory Consumption

(c) LMS Found **(d)** Application Size

Figure 3.4 Static Analysis – The Y-axes are in logarithmic scale

For experimental analysis, we have implemented *DisProTrack*[4] and executed it in a testbed deployed in our lab. The implementation details are as follows.

3.5.1 Experimental Setup

The experiments are executed on a workstation having Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz dual-core processor and 32 GB of memory. To ensure bare metal infrastructural abstraction, we use multiple Virtual Machine (VM) instances running with Ubuntu 22.04 LTS with Kernel version 5.15.0 – 39-generic. The compute functions are deployed using Docker[5] across the Virtual Machines (VMs). We use docker-swarm[6] for container management where one container instance is deployed as the manager. For communication, we use an overlay network driver to ensure direct connectivity among the containers. During our experimentation, we have used the standard docker images of the applications collected from `Dockerhub`[7] with an added image layer of `auditd` as described in section 3.4.2. On the other hand, the proposed static analyzer and runtime engine are deployed as a containerized micro-service.

[4] https://github.com/usatpath01/DisProTrack (Accessed: May 5, 2026)
[5] https://www.docker.com/ (Accessed: May 5, 2026)
[6] https://docs.docker.com/engine/swarm/ (Accessed: May 5, 2026)
[7] https://hub.docker.com/ (Accessed: May 5, 2026)

3.5.2 Resource Utilization

We conducted experiments to analyze the resource utilization and overhead imposed by *DisProTrack*, which has a two-fold view – (i) the overhead of the static analyzer, which is a one-time event for every deployment scenario, and (ii) the overhead of the runtime engine, which is a periodic event. Therefore, we present different sets of experiments to understand these overheads as follows. Each experiment is repeated 10 times to account for nondeterministic behavior caused by container scheduling, network conditions, and concurrent resource contention, and the mean is shown in the plots. Our evaluation is consistent with the existing literature on systems and microservices that have conducted large-scale experiments 10 to 20 times and reported average results due to the cost and unpredictability of distributed system setups.

(i) Overhead of Static analyzer

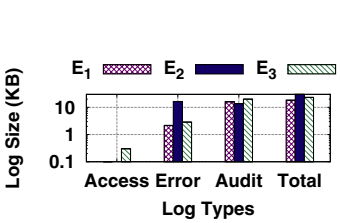
To understand the resource utilization overhead during static analysis, we have considered 7 different applications as given in table 3.1.

Table 3.1 List of Benchmarked Applications

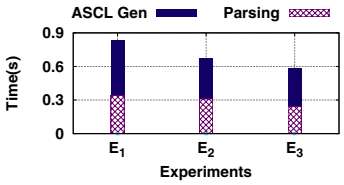
Name	Type of Application
`Apache Webserver`	Together popular as LAMP-Stack and widely used for web service deployment
`PHP-FPM`	
`MySql`	
`DynamoDB`	A noSQL based database used in Amazon Lambda stream processing usecase
`ImageMagick`	An image processing framework can be used for various Amazon Lambda file processing usecases
`OpenVPN`	A popular Secure IP tunnel daemon
`Wget`	Linux network downloader

Based on the obtained results (figs. 3.4a and 3.4b), we observe that the static analysis for `MySql` takes the longest time to complete and needs a greater amount of memory. The time and memory requirements increase when the BT increases. This is justified as the value of BT increases, the number of backtraces also increases (as mentioned in section 3.4.1). However, an increase in the number of backtraces can identify more number of LMSes as shown in fig. 3.4c. We also observe that even though the size of `Magick` is greater than the size of `MySql` (fig. 3.4d), it takes more time and memory for `MySql` to trace the LMSes. This is due to the nature of the program control instructions used by developers while developing the ap-

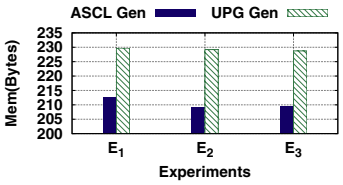
plication. Hence, it takes more time and memory to complete the backtrace in the presence of higher number of indirect branch instructions. For the same reason, `PHP-FPM` takes less amount of time and memory than `MySql`; however, it identifies more number of LMSes.



(a) Log File Size

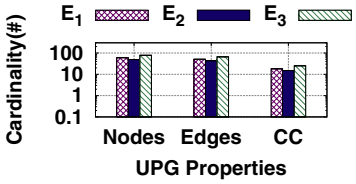


(b) Phase wise execution time

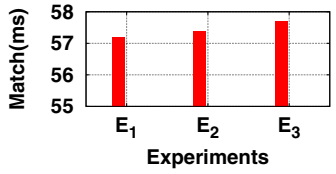


(c) Phase wise Memory utilization

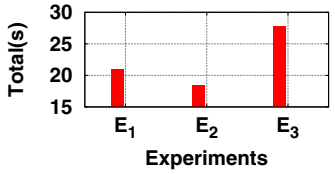
Figure 3.5 Runtime Analysis – The Y-axis in (a) is in logarithmic scale



(a) UPG Parameters



(b) RegEx Identification Time



(c) UPG Construction time

Figure 3.6 Runtime Analysis – The Y-axis in (a) is in logarithmic scale

(ii) Overhead of Runtime Engine

Since the runtime engine works for a pipeline of micro-services, we execute multiple experiments on a web service application. Our deployed LAMP stack web service is composed of 3 micro-services (similar to fig. 3.2); (a) [μ_1]:`Apache` based web server, (b) [μ_2]: `PHP-FHM` for server-side request handling, and (c) [μ_3]: `MySQL` for handling queries generated by the `PHP-FHM`. Each HTTP request incident on μ_1 forwards a `FastCGI` requests to μ_2. μ_2 executes the handler functions and accesses the database deployed in μ_3 for dynamic content. Once the page is constructed, μ_1 returns it as a response to the client. Specifically, we have hosted an application authorization portal where μ_1 interacts with the μ_2 via μ_3 to validate the user credentials. Upon receiving the valid credentials, μ_1 redirects from the login page to a user-specific home page. Otherwise, it remains on the same page with an error message pop-up. On top of the authorization service, we consider three experimental login scenarios as follows.

E_1 New users register themselves, and the details are updated in the database. Once registration is successful, the user tries to log in and then log out of the application with valid credentials.

E_2 A user logs in with a valid credential and goes to the home page. Once logged in, they try to reset the password and re-login with a new password.

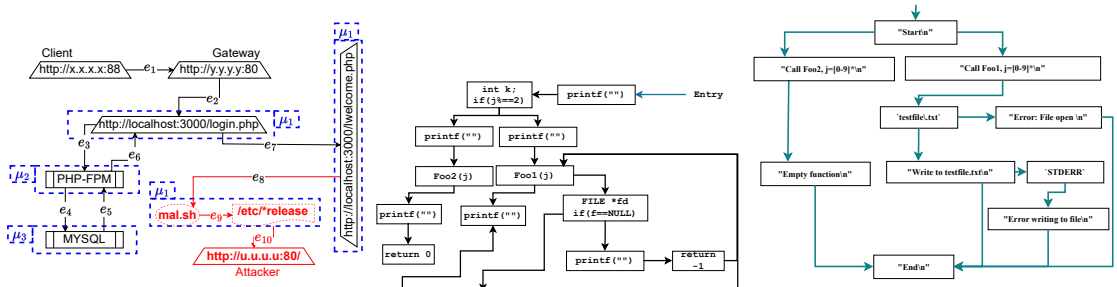
E_3 A user tries to log in to the deployed web service. On the homepage, they click on a link that triggers a background script and redirects the user to a different IP.

The size and types of log files generated during these experiments are presented in fig. 3.5a. The results show that the error logs generated during E_2 are more significant than E_3. In contrary, the access log generated by E_3 is much greater than the rest of the two scenarios. However, the audit log generated by E_3 is greater than the other two. The log file size depends on a user's actions while browsing the web service.

Next, we present the phase-wise time consumption analysis in fig. 3.5b. Here we observe that the time required to parse the log files, merging them to create an ASCL and generation of UPG during provenance builder (shown in fig. 3.6c) is directly related to the total amount of logs generated during the experiments which take approximately 600ms-900ms to complete the runtime processing. After contextualizing the log files, the system administrator provided suspicious log messages converted into an equivalent regular expression (RegEx) to avoid the values of the variables. This generated RegEx string is searched across the LMS-CFG which takes only a few milliseconds (as shown in fig. 3.6b). We also provide the memory consumption during the individual phases as presented in fig. 3.5c which suggests that the memory footprint is significantly lower for the modules of the runtime engine (in between 200B to 250B).

We observe that depending on the scenarios, the nature of the UPG is different, as shown in fig. 3.6a. We find that E_3 has a significantly higher amount of nodes and edges than the rest of the two cases. The number of connected components in the UPG is considerably higher for E_3. Each connected component in the graph represents an execution unit of a process, which helps resolve the dependency explosion problem. Only the related events of a process form a connected component. More number of connected components implies that the graph is more densely partitioned.

3.6 Analysis



(a) UPG for Confidential Data Theft **(b)** CFG for Test Program **(c)** LMS-CFG for Test Program

Figure 3.7 A PoC Case Study to Analyze the Accuracy of *DisProTrack*

This section discusses the effectiveness analysis of *DisProTrack*. In this context, we consider an adversarial scenario. The static analyzer can easily detect an adversary who can modify the application's source code. However, an adversary accessing the runtime platform can evade the static analysis and may issue malicious operations during code execution. Therefore, in this section, we primarily focus on detecting runtime adversaries. We have simulated multiple attack scenarios by considering different adversary models. We next discuss one of them.

3.6.1 Adversarial model & Attack Scenario

We assume that an adversary can somehow bypass the authorization mechanisms and gain access to one/more application container(s) except the runtime engine container without being detected. In the compromised container(s), the adversary can add, modify, and/or execute scripts and deploy webpages. However, We assume that the logs and audit rules are part of the Trusted Computing Base (TCB). Moreover, the communication between the compromised container and the runtime engine can not be adulterated.

Using this adversarial model, the attacker may perform different types of malicious activities. However, here we present the case study in light of *confidential data theft* attack[8]

[8] https://bit.ly/trendmicro_shading_light (Accessed: May 5, 2026)

where the attacker attempts to insert a malicious script to steal the confidential information from the compromised system. This script can be triggered during execution time. We simulated the attack on top of our web service application described in section 3.5.2 by placing a malicious script named *mal.sh* in `PHP-FPM` container. This script is executed when an authorized entity login to the website after successful authentication and clicks on a masked link on the welcome webpage. Once the script gets triggered in the background, alongside normal execution, it tries to access and read a sensitive file on the server and forward them to an attacker's server IP address.

```c
#include<unistd.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#define Foo2(int j) ({printf("Empty function\n");})

int Foo1(int j){
    FILE *f = fopen("testfile.txt", "a");
    if (f == NULL) {
        printf("Error: File open\n");return -1;
    }
    if (fprintf(f, "Write to testfile.txt\n") < 23) {
        fprintf(stderr, "Error writing to file\n");
        return -1;
    }
    return(0);
}
/**********************/
int main(void){
    printf("Start\n");int j = rand();
    if(j%2==0){
      printf("Call Foo1(), j = %d\n", j);Foo1(j);
    }else{
      printf("Call Foo2(), j = %d\n", j);Foo2(j);
    }
    printf("End\n");
    return 0;
}
```

Listing 3.1 PoC Program for Accuracy Analysis

3.6.2 Provenance Builder for Attack Detection

Let us understand how this attack can be detected using *DisProTrack*. The UPG constructed by *DisProTrack* for above attack scenario is presented in fig. 3.7a. To ensure readability, we have omitted a few UPG metadata and masked IP addresses. Using the relative event sequence numbers/edge identifier, the sequence of events can be traced in order. From the particular UPG instance, we can find that a client with IP address $x.x.x.x$ is connected to the service via port 88. The client takes the normal authentication route (from e_1 to e_7) to reach the `welcome.php` page. After that, the `mal.sh` script is triggered which results in collection of data from `/etc/*release` directory and forward it to $u.u.u.u$ (from e_8 to e_{10}). This step is a deviation from the standard behavior; therefore, the system administrator must intervene in this case and take some preventive action (e.g., suspend user access, block IP, etc.).

3.6.3 Accuracy analysis of *DisProTrack*

During our development of *DisProTrack* and experimentation with several other attack scenarios, we observed that the detection of attack depends on the accuracy of LMS-CFG construction from CFG in the static analyzer. Although we have presented the LMS identification results in fig. 3.4c for different applications, the lack of gold standard values restricts us from claiming the accuracy of the proposed algorithm 1. Therefore, to justify the accuracy of the static analyzer, we present a small sample proof-of-concept (PoC) program (listing 3.1) here. The sample program presents two function calls depending on a random number generated. The functions can either generate a log message in the `STDERR` console or in a log file. As the program is simplistic in nature, it is easy to ascertain the accuracy of the generated LMS-CFG from the framework presented in fig. 3.7c. For comparison purposes, we present the corresponding CFG in fig. 3.7b, which is also obtained from the static analyzer module. From these two figures, we observe that both the figures have 8 listed LMSes and two file handles. The causal paths among the nodes are also verified to ensure that all the paths are covered in the corresponding LMS-CFG.

3.7 Summary

In this chapter, we developed a non-invasive causality analysis system named *DisProTrack*, tailored for distributed serverless applications to facilitate effective provenance tracking and root cause analysis. *DisProTrack* automatically constructs a Universal Provenance Graph (UPG) by combining container-level application logs and system audit logs without requiring any instrumentation of the application source code. Our investigation revealed that existing solutions fail to scale for serverless environments due to heterogeneous log formats, distributed microservice execution, and lack of correlation across system and application logs. To address this, we introduced a two-phase framework consisting of a static analyzer and a runtime engine. The static analyzer processes the application binaries to extract Log Message Strings and their control flow graphs (LMS-CFG), while the runtime engine intercepts write system calls using a Loadable Kernel Module (LKM) to attach unique identifiers to logs and periodically aggregates them. *DisProTrack* then constructs execution units and interlinks them to form the UPG using regular expression matching and syscall context. We ensured lightweight deployment as a microservice with negligible resource overhead. Real-world experiments were conducted on several benchmarked serverless applications, including LAMP stack and others, showing the framework's efficiency, minimal memory footprint, and its utility in detecting advanced attack scenarios like confidential data theft. Notably, *DisProTrack* achieves accurate execution trace reconstruction within 20–30 seconds. The current method is limited to the deployability as it leverages the Linux Kernel Module (LKM). In the following chapter, we developed a system to overcome the deployability issues associated with LKM that has a significant dependency on the Linux kernel versions.

Chapter 4

A Observability Framework for Distributed Sandboxed Microservices

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 track-

[1] https://www.cncf.io/ (Accessed: May 5, 2026)

ing predefined metrics, collecting logs, and identifying symptoms of operational issues, observability [9, 32] 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.

Also, the general-purpose system call telemetry systems such as Sysdig [64], SysFlow [62], and eAudit [63] provide valuable system-level observability, but lack in providing distributed causality and cross-layer context bridging (Application-layer and OS-layer). While [64] offers rich per-host syscall tracing with container awareness, it lacks inter-host ordering guarantees and relies on simple timestamps. [62] reduces log volume by summarizing syscall data into flow records, but at the cost of discarding fine-grained causal relations, making deep debugging and root cause analysis difficult in multi-host environments. Although [63] also supports full syscall monitoring, its scope is limited to single-host, OS-level auditing and lacks container/cloud-native awareness. These limitations motivate the development of an observability framework that preserves both intra- and inter-host causality and enables 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 [111, 112]. 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 [103, 113] dynamically, which can further be used for system vulnerability detection [46,114], root-cause analysis [115–117], etc. Our research work 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.* 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` [9, 118, 119] 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 [120, 121]. (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 chapter 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 chapter 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 Extended Berkeley Packet Filter (eBPF) [122–125] 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` [126] social media application with 30 microservices, each with 3-5 replicas, spawning over 10-30 different hosts, and compare its performance with `Tracee` [40], 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.

4.1 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*.

4.1.1 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 $L(e_i), L(e_j) \in \mathcal{L}$,

$$e_i \prec e_j \iff L(e_i) \mapsto L(e_j), \tag{4.1}$$

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.

4.1.2 Basic Working Principle and System Overview

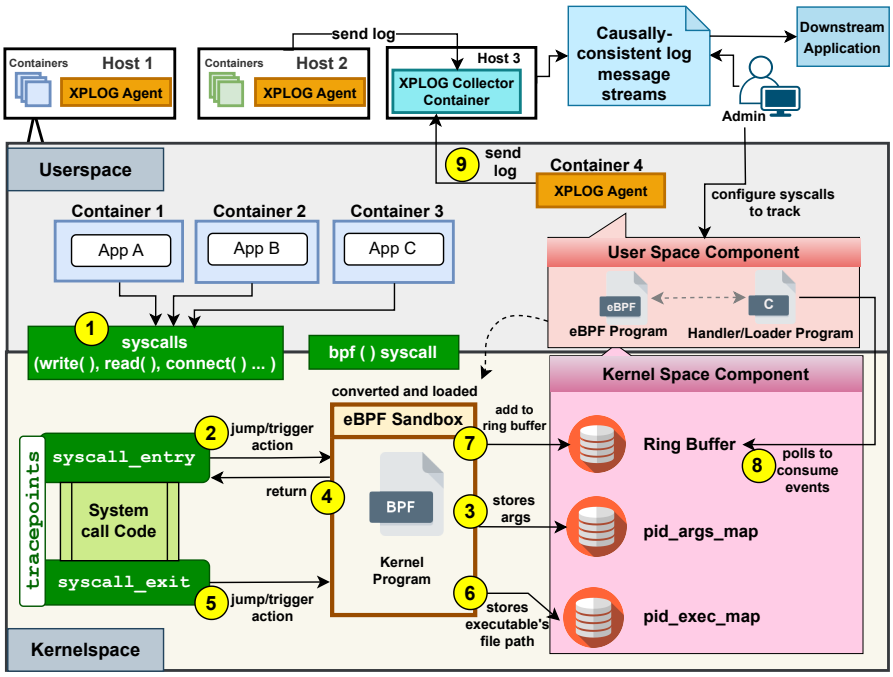


Figure 4.1 *XPLOG*: User and Kernel-space Components

Following the general principle of microservice architecture [127, 128], 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 4.1) 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 4.1 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 up to 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* [129]) 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 in 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 fig. 4.1, *XPLOG* comprises (1) the *XPLOG Agent*, resides within each host (i.e., multiple physical machines) and is 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.

4.2 Components of XPLOG

The *XPLOG* Agents are deployed as privileged containers over each host/physical machines, which allows them to monitor the microservice logs (application-level logs) from other containers. Also it requires capabilities `CAP_PERFMON`, `CAP_SYS_ADMIN`, and `CAP_BPF` to attach eBPF probes. However, the monitored microservice containers are not privileged containers and do not need to be modified in any way. The user-space and the kernel-space components of *XPLOG* Agents work as follows.

4.2.1 *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 4.1. Although the probe creation and other auxiliary activities are explained in section 4.2.2 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.

4.2.2 *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.

(i) 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.

(ii) eBPF Maps

The maps are used to share data between the user and the kernel-space. We use following eBPF maps:
(a) *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*.

(b) *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).

(iii) 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 sin-

gle 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. 4.2). 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 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.

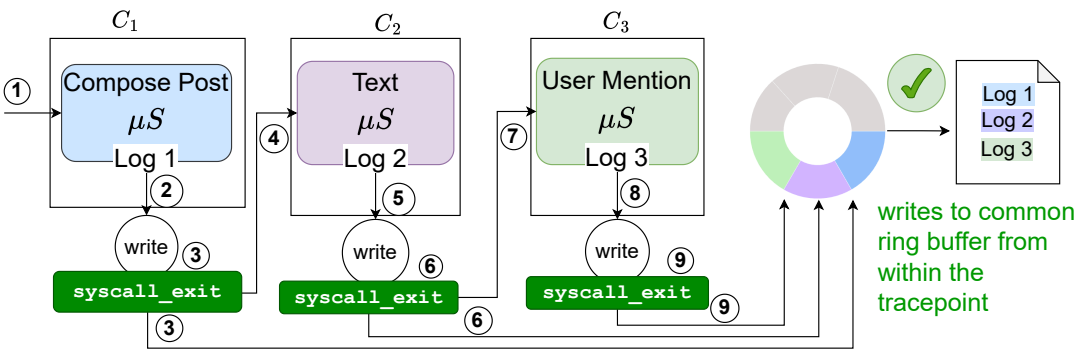


Figure 4.2 Using eBPF ring buffer to ensure causal ordering while generating a common log file

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 4.1). 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.

Selective Logging Filter: Because the volume of system call events generated during the running of containerized microservices is large, *XPLOG* employs filtered logging at the user space by the *XPLOG* Agent and at the kernel using eBPF. Rather than logging all instances of monitored system calls, *XPLOG* identifies and logs only those events that are either causally or semantically important to the application's activities. Specifically for file I/O syscalls such as write(), *XPLOG* analyzes the file descriptor to determine if the output stream is standard output (e.g., `STDOUT, STDERR`) or noted application logging destinations which include files situated in `/var/log/*`, or files ending with `.log`. These are classified as application logs and are integrated into the log stream. For system events, *XPLOG* only keeps syscalls that affect process execution state (e.g., execve, clone, exit) or system-wide resources like file and socket interactions (open, connect, accept, read, write), if in a causally connected request (detected through tagID propagation).

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. 4.1 ①), which triggers a syscall entry tracepoint program (fig. 4.1 ②) in the *XPLOG* Agent. This program populates the `pid_args_map` with the host PID & Thread ID and the syscall arguments (fig. 4.1 ③). Then the system call executes (fig. 4.1 ④), and once the execution is complete, it triggers the syscall exit tracepoint (fig. 4.1 ⑤) to populate the executable file path of the event to the `pid_exec_map` (fig. 4.1 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. 4.1 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

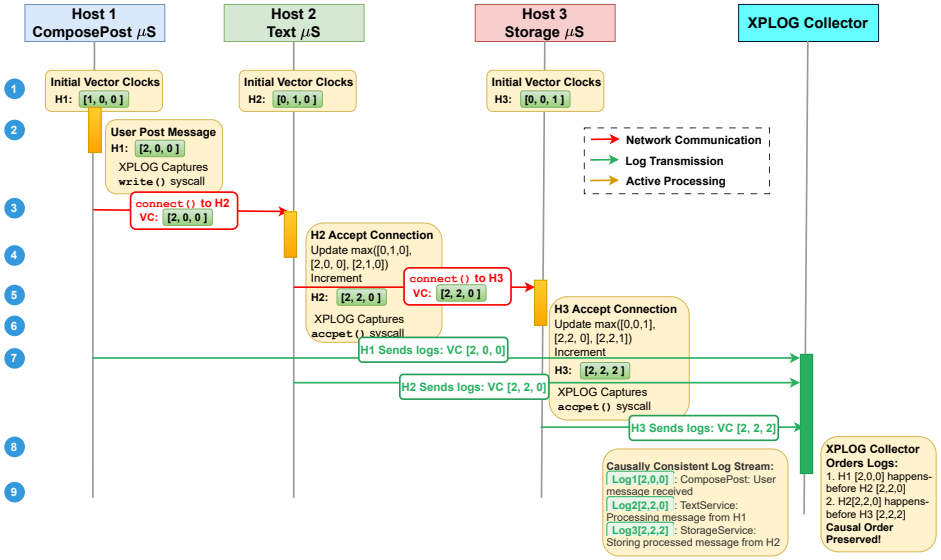


Figure 4.3 Three hosts H1(ComposePost), H2(Text), H3(Storage) processing an user request. Initial State: H1[1,0,0], H2[0,1,0], H3[0,0,1]. An example showing how vector clocks are updated during inter-host communication in a three-microservice deployment.

polls the ring buffer (fig. 4.1 ⑧) and writes these atomic log entries to the *XPLOG* Collector section 4.2.3 (fig. 4.1 ⑨). The steps between (fig. 4.1 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 host. Also, to preserve per-thread causal ordering on multi-core systems, *XPLOG* maintains a monotonic local logical [130] counter at the entry probe of each syscall. This counter is attached to the log entry and incremented per thread context. This counter is 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.

4.2.3 *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*" [129] implemented in the kernel space triggered with each cross-host communica-

tion event. Figure 4.3 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 write 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 the vector clock of H_2 according to the merge rule: $VC[i] = max(local\_VC[i], received\_VC[i])\forall i$, 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 $H3$ (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 local component increment. 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 ($VC_1 < VC_2 \iff VC_1[i] \leq VC_2[i]\ \forall i$ and $\exists j : VC_1[j] < VC_2[j]$) 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. 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 [129]. We are not modifying 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.

4.3 Performance Evaluation

We evaluate *XPLOG* with the following objectives: (1) how well *XPLOG* can reduce disorders in the generated log compared to `Tracee` [40], 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`.

4.3.1 Implementation Details

Table 4.2 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*

The source code, documentation, and the experimental configuration scripts of *XPLOG* implementation have been open-sourced[2]. The implementation of *XPLOG* Agent consists of 3232 lines of code (LoC) and *XPLOG* Collector is 176 LoC. 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 4.2), 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 4.1. 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 4.2.2) 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

[2] https://github.com/usatpath01/Pluggable_Logging (Accessed: May 5, 2026)

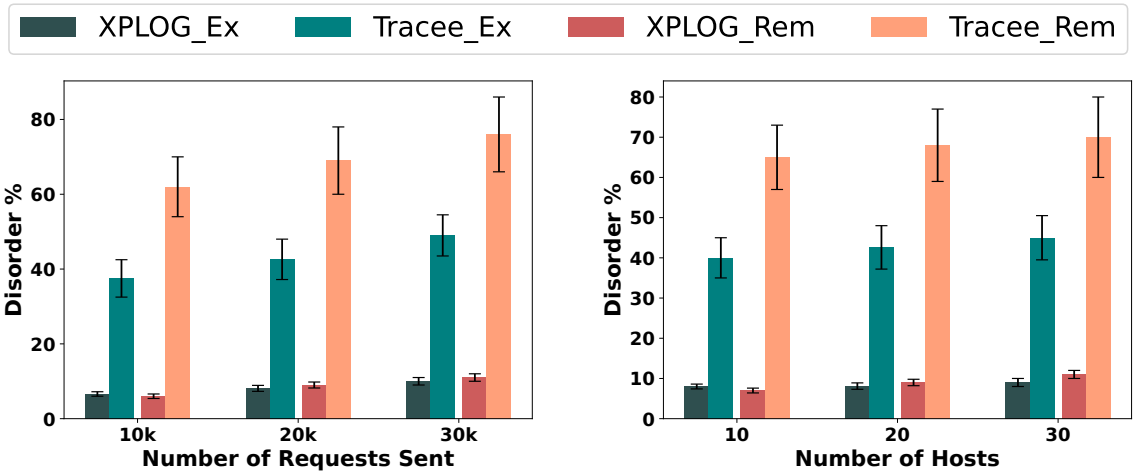
XPLOG collector. Additionally, we have reserved 2MB for `pid_exec_map` and 64KB for `pid_args_map`.

4.3.2 Experimental Setup

To evaluate the efficacy of *XPLOG*, we use DeathStarBench [126], 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 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* and also to assess scalability, 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. This allows for the testing of cross-host causal ordering, log collection overhead, and scalability under increasing deployment sizes. The goal here is not to replicate the hyperscaler cloud providers but to test a representative enterprise-scale production environment where microservices are deployed across multiple nodes with replication. 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

[3] https://github.com/delimitrou/DeathStarBench/raw/master/socialNetwork/figures/socialNet_arch.png (Accessed: May 5, 2026)

[4] https://github.com/wg/wrk (Accessed: May 5, 2026)



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

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

Figure 4.4 Log disorder between `Tracee` and *XPLOG* (Service endpoint: CP+UT+HT)

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. However, *XPLOG* can 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` [40] 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.

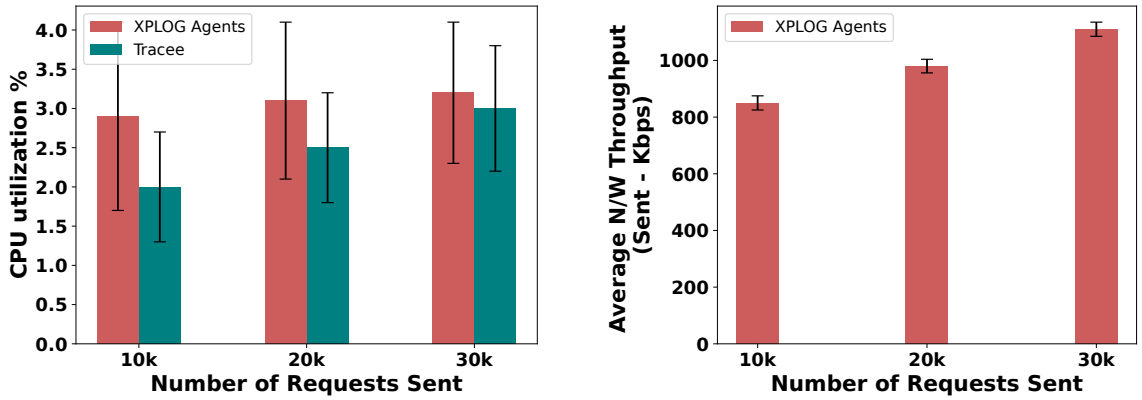
4.3.3 Analysis of Causal Ordering Level

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 se-

quential 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 $(10,000$-$30,000)$ 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 [131]: (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.

Figure 4.4(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. 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 4.3.5, table 4.4). Similarly, fig. 4.4(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 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.



(a) Avg. CPU Utilization

(b) Avg. Netw. Traffic to Collector

Figure 4.5 Total 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.

4.3.4 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 4.3. 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 4.3, 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.

The filtering, although intended to be used in runtime observability and real-time detection, can also be used in offline forensic analysis. This is because, by reconstructing the causality links from archived logs and vector clock data, irrelevant data can be filtered out before constructing the graph.

Table 4.3 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

4.3.5 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.

[5] https://github.com/google/cadvisor (Accessed: May 5, 2026)
[6] https://github.com/prometheus/node_exporter (Accessed: May 5, 2026)

(i) CPU and Network Utilization

Figure 4.5 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. 4.5a) of *XPLOG* Agents are 2.9%, 3.10%, and 3.15% when $10K$, $20K$, and $30K$ requests are sent concurrently. 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. 4.6, 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 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. 4.5b) of the *XPLOG* agents, which is $\approx 1.7Mbps$, because of the constant streaming of the generated log messages to the collector.

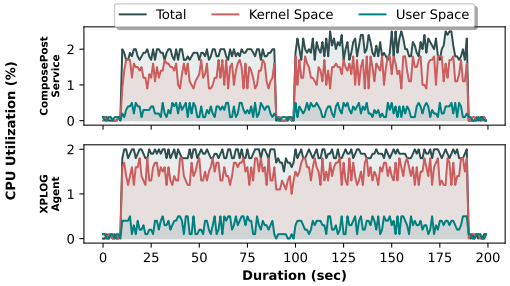
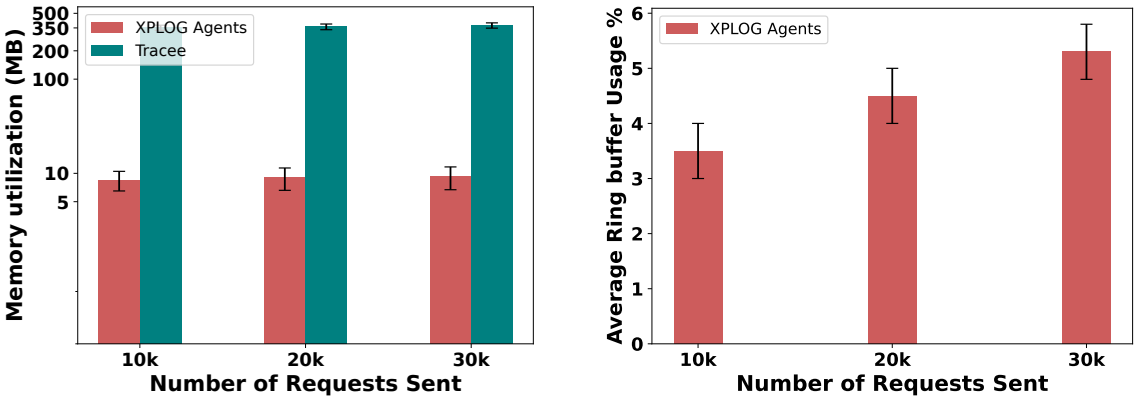


Figure 4.6 CPU utilization per request (Service endpoint: CP)

(ii) Memory Overhead

We observe that the memory overhead (see fig. 4.7a) 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 4.4). Notably, *XPLOG*



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

Figure 4.7 Average Memory overhead and eBPF buffer usage of 19 *XPLOG* Agents

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 4.4.

XPLOG can handle this high amount of logs reasonably well, which can be justified by fig. 4.7a 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 4.3), 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 4.3.4), resulting in meaningful log entries in the collated log message stream.

Table 4.4 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

The percentage utilization of the eBPF ring buffer while varying the number of requests from $10K$ to $30K$ is presented in fig. 4.7b. We have declared the eBPF ring buffer with __uint(max_entries, 256 * 1024), which specifies the total size of the buffer in bytes. This

means the ring buffer has a total capacity of 256 * 1024 = 262144 bytes (≈ 256 KB). Each log entry in our case (Refer Listing 1) is approximately 500 to 600 bytes. Based on this size, the buffer can accommodate up to around 437 log entries before it becomes full. The 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.

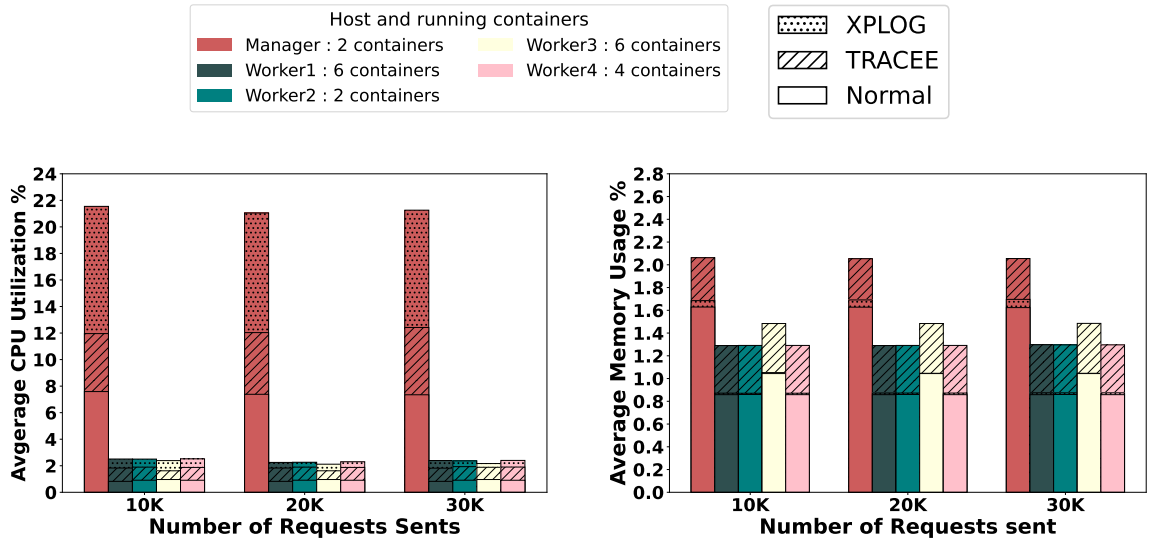
Table 4.5 Log writing latency (#Concurrent requests: $10,000$)

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

(iii) 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. 4.9). 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. 4.9a). However, with *XPLOG*, it goes up to 21% (shown in dotted of fig. 4.9) as the collector continuously consumes the log messages sent by the agents. However, we observe that the memory usage (shown in fig. 4.9b) 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 [132, 133]. 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.



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

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

Figure 4.9 Host Resource utilization (Service endpoint: CP+UT+HT)

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 4.5) 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.

4.3.6 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 4.1 with relevant fields highlighted for ease of understanding. Each log entry contains fields as described in table 4.1. [7].

[7]Sample logs for *XPLOG* and Tracee are available at https://github.com/usatpath01/Pluggable_Logging/tree/main/Logs (Accessed: May 5, 2026)

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

Listing 4.1 Example of *XPLOG*-generated log entries

((i) Ease of Log Consultation

As shown in table 4.4, *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 4.1). 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 4.1 (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 4.1).

(ii) 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 4.2 and collected the logs while executing the application above using both the frameworks.

Table 4.6 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	✗	✓

We observe that *XPLOG*-generated application and system logs (listing 4.1) provide more information in comparison to the `Tracee`-generated logs (listing 4.2). A summary of the comparison (table 4.6) 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 4.2). Moreover, `Tracee` provides only the file descriptors value, where *XPLOG* provides more readable path information (see line 16 of listing 4.1). Additionally, *XPLOG* captures the syscall arguments, which provide several additional information, like the network addresses from socket calls (see Line 22 of listing 4.1), 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 (Accessed: May 5, 2026)

```
TIME     UID  COMM             PID     TID     RET     EVENT                    ARGS
21:20:32:498568 0   PostStorageServ    1       1       0       security_socket_bind
    sockfd: 10, local_addr: map[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, ...
21:20:34:435046 0   ComposePostServ    1       1       0       security_socket_accept
    sockfd: 8, local_addr: map[0.0.0.0 sin_port:9090]
21:20:34:448360 0   TextService        1       1       0       security_socket_accept
    sockfd: 8, local_addr: map[0.0.0.0 sin_port:9090]

```

Listing 4.2 Tracee-generated syscall log snippet

4.4 Summary

In this chapter, we developed a dynamic observability framework named *XPLOG*, specifically designed for distributed sandboxed microservice environments. *XPLOG* enables real-time, causally consistent log generation without requiring application instrumentation or OS kernel modification. Our investigation reveals that preserving the causal ordering of system and application logs is crucial for effective provenance tracking and root cause analysis in distributed systems. *XPLOG* leverages eBPF-based probes to intercept system calls and aggregates both system and application-level logs while maintaining execution order. It employs a grey-box approach to monitor containerized applications and dynamically maps application logs to corresponding syscalls, thereby ensuring contextual log collation. *XPLOG* uses eBPF ring buffers and vector clocks to preserve intra- and inter-host causal consistency across logs. Importantly, *XPLOG* enriches logs with detailed metadata including task context, syscall arguments, and artifacts, facilitating deeper forensic analysis. Extensive evaluation using the DeathStarBench social network application shows that *XPLOG* significantly reduces log disorder and improves causal consistency compared to Tracee. *XPLOG* achieves this with minimal CPU, memory, and network overhead, making it suitable for edge computing scenarios. We further demonstrate *XPLOG*'s effectiveness in filtering relevant logs for event-based queries, reducing query time and eliminating irrelevant log entries. *XPLOG*'s expressive and unified log format enables system administrators to easily trace request pathways across distributed microservices. In the next chapter, we explore how these enriched causally-ordered logs can be transformed into dynamic provenance graphs for attack detection using classical supervised machine learning models.

Chapter 5

Supervised Attack Investigation in Microservices Using Provenance Graphs

Modern large-scale distributed applications rely on microservice-based architecture [134, 135] due to its flexibility, scalability, and modular deployment, thus offering usability, robustness, and fault tolerance from the service management perspective. However, such architectural paradigm also introduces new attack surfaces [136–138], enabling sophisticated attack vectors, such as *Advanced Persistent Threats* (APTs) [11, 12] where a group of attackers can conduct large-scale targeted intrusion by exploiting the distributed and loosely-coupled nature of microservices, thus launching stealthy multi-step attacks that traverse multiple services and hosts, often remaining undetected for extended periods [12]. Traditional security approaches fail to detect such attacks due to their inability to correlate low-level system events across the hosting environments and the higher-level application activities across a distributed environment. Therefore, developing an observability framework across distributed microservices is essential to detect and investigate such attack vectors in real-time.

The classical approaches for runtime attack investigation use system provenance graphs [43, 46,47,50,52,89,107,114], where a dynamic graph structure captures the correlation across various system and application events, indicating the flow of execution of the underlying application. Sophisticated machine and deep learning (ML/DL) techniques [49–51,

107] have been developed to identify possible attack vectors over a runtime provenance graph. However, such existing methods of attack investigation fall short for distributed microservices-based applications because of the following reasons.

• Reason ①: *Correlating logs across hosts.* – Microservices may span multiple hosts; thus, a single request can trigger a chain of interactions across several hosts. However, the existing logging framework treats every host independently and in silos. Therefore, generating runtime provenance becomes difficult as the logs from individual hosts need to be analyzed manually to correlate various application and system-level events.

• Reason ②: *Semantic gap between different layers of logs.* – APTs are multi-stage attacks that typically span across both the application as well as the infrastructure. Thus, the detection of APTs needs to extract the correlation across the application-generated events and the system (OS-level) events. Notably, microservice-based applications use various levels of abstraction and sandboxing, involving OS-level virtualization (virtual machines), software virtualization (containers), and service virtualization (web assembly). The existing logging tools follow different semantics at different sandboxing layers. For example, containers use a separate process ID (PID) namespace; thus, the PID of a containerized application may be the same as the PID of a system-level process. Consequently, correlating the events from different namespace hierarchies becomes challenging when constructing the runtime provenance graph.

• Reason ③: *Lack of benchmarking events and datasets.* – Given that existing logging mechanisms fall short of capturing the correlation across various application and system events during APTs, there is a lack of proper benchmarking methods and datasets to build up a generic robust model for APT investigation and attack detection. This particularly limits the existing literature on provenance-based analysis as the core of such solutions, i.e., generating the provenance graph, needs a revisit. Towards this, applications often lack the complexity to simulate multi-stage attacks exploiting existing standards such as CVEs (*Common Vulnerabilities and Exposures*[1]) and CWEs (*Common Weakness Enumerations*[2]), limiting the evaluation of detection systems in realistic, vulnerable environments.

[1] https://www.cve.org/ (Accessed: May 5, 2026)
[2] https://cwe.mitre.org/ (Accessed: May 5, 2026)

Considering these challenges, we argue that designing a sophisticated, robust, lightweight runtime attack detection for distributed sandboxed microservices-based applications first needs a method to generate a scalable, environment-sensitive, and dynamic provenance graph architecture by correlating the application and system-level events generated over multiple hosts. As existing logging mechanisms fail to capture such dependency and correlation, we need an inclusive observability framework to dynamically capture and filter out the essential event information from the sandboxed microservices and the host platforms and then construct the runtime provenance graph. Considering such requirements, in this chapter, we present *μProv*, an inclusive observability and provenance framework that captures the dependency and correlation from application and system events across various hosts and then generates the runtime provenance graph dynamically on the fly. With a proof of concept (PoC) microservice-based application and state-of-the-art ML/DL-based methods for attack investigation over provenance graph-based architecture, we show that *μProv* can detect attacks more accurately and faster compared to the scenario when existing logs are used to generate a provenance graph. In summary, our contributions to this chapter are as follows.

• Contribution (1): *Custom eBPF-based logging solution.* – The core of *μProv* uses a novel, low-overhead logging solution based on the *extended Berkeley Packet Filters* (eBPF) [38, 139], designed explicitly for distributed microservice environments. This tool enhances the granularity and efficiency of system call monitoring in complex, containerized applications and helps correlate the application-generated event logs with the system logs. Although existing solutions like [60] have shown the efficacy of eBPF to observe microservices, they are limited to extracting various performance metrics. In contrast, we develop a holistic solution using eBPF to capture correlations across application and system events that are useful for dynamic runtime provenance graph generation.

• Contribution (2): *Extracting dynamic provenance graphs* – *μProv* leverages provenance graphs constructed from low-level system events to detect vulnerabilities while effectively illustrating the causal relationships between processes, file accesses, and network activities, providing a holistic view of system behavior. By tracing the lineage of events, *μProv* can identify anomalous patterns indicative of multi-stage APT attacks, thereby revealing attack sequences that conventional methods may overlook.

• Contribution ③: *Vulnerability integration in microservices and dataset generation* – To tackle the issue of inadequate datasets, we have created a hybrid dataset that integrates real-world microservice logs with synthetically injected attack patterns. For this purpose, we have developed "*PicShare*", a PoC microservice web application that enables users to upload, view, and receive picture recommendations. We have designed an emulation framework to benchmark APTs over microservice-based applications by integrating the PoC application with vulnerabilities drawn from CVEs and CWEs, which can help not only to evaluate *μProv* but also provide a framework for the research community to emulate sophisticated attacks over large-scale microservice-based architecture.

• Contribution ④: *Empirical evaluation & performance-accuracy trade-off analysis* – We empirically analyze various attack scenarios, identifying key indicators of microservice attacks. This analysis highlights the main factors distinguishing normal behavior from attacks while revealing the current limitations of detection methods. We compare *μProv*'s performance with *Tracee*, an eBPF-based logging framework, in generating system provenance to detect microservice-based attacks. We observe that *μProv* can help classical ML-based attack investigation methods detect attacks over distributed microservices with better accuracy and granularity while providing a scalable and real-time framework for attack investigation.

5.1 Design Goals & System Overview

Consider a distributed photo-sharing application built on a microservices architecture. A user action, such as photo uploading, triggers multiple interactions between services like user authentication, access control, and database management. Typically, these services generate specific sequences of system calls and events recorded in the logs. However, an attacker might exploit a vulnerability in the upload service by injecting a malicious PHP script that performs *Remote Code Execution* (RCE) [140]. Notably, when analyzed in isolation, this attack could appear as part of the normal flow. However, by correlating logs across multiple hosts, the system may be able to identify irregularities in system call sequences. Our system architecture must meet the following design requirements to detect such multi-stage attacks within distributed microservice environments.

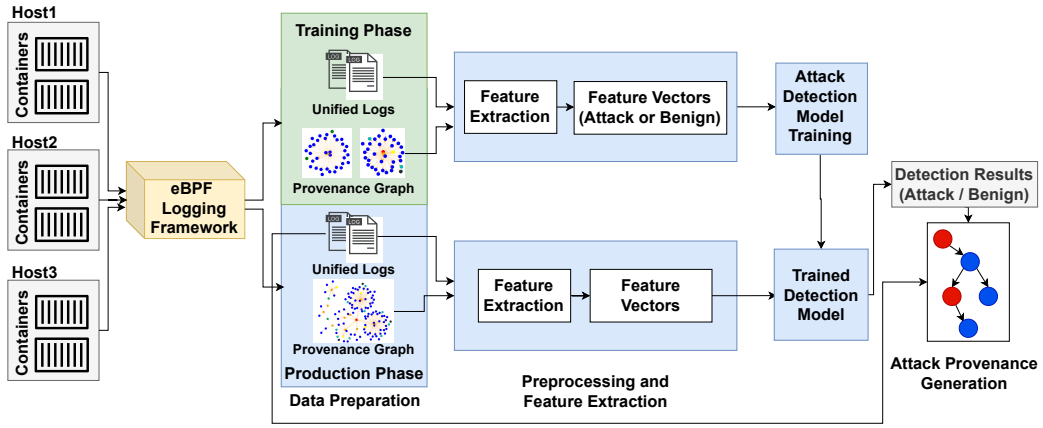


Figure 5.1 *μProv*'s Architecture

$\mathbf{G_1}$: To design and deploy a logging module within the host OS kernel that operates in an application-agnostic manner and generates a causally consistent stream of log messages across the hosts in distributed systems. The *causally-consistent log messages* [59] should be able to capture the dependency across various system and application events by connecting and correlating them across the hosts.

$\mathbf{G_2}$: Our system should enable dynamic and incremental creation of provenance graphs that capture the causal relationships between system calls and events.

$\mathbf{G_3}$: Our framework should enable real-time detection of anomalous activities using machine learning models trained on system call provenance graphs.

As illustrated in fig. 5.1, *μProv* consists of three key components to achieve the above design goals: (1) the *Logging Framework*, deployed on each host, responsible for ensuring the causal consistency of log messages generated by microservices and aggregating these logs across hosts while preserving causal relationships; (2) the *Provenance Graph Generator* that dynamically constructs provenance graphs from the collected logs; and (3) the *Machine Learning Module* that analyzes the graphs in real-time to detect potential attacks.

5.2 *µProv*'s Components

We next discuss various components of *µProv* in detail.

5.2.1 eBPF-based Logging Framework

We propose a logging framework leveraging eBPF [139] that collects system and application logs to generate a unified causally ordered log from different microservices running in a distributed architecture. In this section, we briefly discuss the key components of the framework, including log collection, preprocessing, and causal ordering mechanisms. Our Logging framework leverages `eBPF` to deploy its logging module directly within the host OS kernel, as shown in the fig. 5.2, in an application-independent manner, producing a causally-ordered global log file, $\mathcal{L}$. Using `eBPF`, custom code known as *probes* can be injected into specific kernel hook points, called *tracepoints*. This allows the kernel's capabilities to be extended safely and efficiently at runtime without modifying the kernel source code or loading kernel modules. As illustrated in fig. 5.2, our system consists of two main components:

(1) The Logging Container – Which runs on each host to ensure causal consistency among the log messages generated by microservices running on that host.

(2) The Log Collector Container – Which gathers log messages from multiple hosts while preserving causal relationships. The Collector subsequently streams these log messages to the *Provenance Graph Generator*, which incrementally and dynamically creates the provenance graph, as outlined in Algorithm 3 & 4 (discussed in the subsequent subsection).

The proposed logging framework monitors the system-level events using *eBPF* probes on system call entry and exit tracepoints, thus ensuring that all the relevant logs corresponding to a syscall event are generated simultaneously, therefore preserving the causal context for a microservice running over

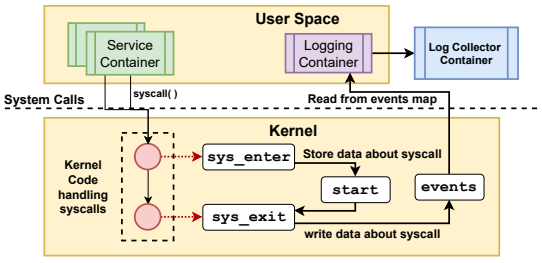


Figure 5.2 eBPF-based Logging Framework

```
1  /* Application Log */
2  Host 1 : {"event_context": {"ts": 173989395151310,"datetime": "09:00:51","
       task_context": {"host_pid": 314463, "host_tid": 317863,"task_command" : "
       PhotoService" ··· }},
3      "data": {"lms" : "Photo Service Response: {\n  filename: '1762071b-9466-48ef
       -87ae-cd956b34c4fb.png',\n  message: 'File uploaded successfully',\n  status:
        'success'\n}\n"},
4      "artifacts":{"exe":"/usr/local/bin/PhotoService"}}
5
6  Host 2 : { ··· "task_context" : {"host_pid" : 1532108,"host_tid":1535513, ···}, "data"
        : { "lms" : INFO:app:User user99 registered successfully\n }, "artifacts":{"
       exe":"/usr/local/bin/UserService"}},
7  /* System Log */
8  Host 1 : {"event_context": {"ts":XXX,"datetime":"XXX", "syscall_id": 42,"
       syscall_name": "connect",
9          "task_context": {"host_pid": 314463, "host_tid": 317863, ···}},
10     "arguments":{"uservaddr":"0x80003","addrlen":16},
11     "artifacts": {"exe": "/usr/local/bin/PhotoService,"IP":"10.0.2.22","port
       ":"34835"}}
12 Host 1 : { ··· "syscall_name" : "read", "task_context" : { "host_pid" :  314463, "
       host_tid" :  317863, ··· "artifacts" : { "exe" : "/usr/local/bin/UserService",
        "file_read" : "/var/lib/docker/containers/···/resolv.conf" }}}, ···
13 Host 1 : {···"syscall_name":"send","task_context":{"host_pid":314463, "host_tid
       ":317863,···}}
14 Host 2 : {···"syscall_name":"recv", "task_context":{"host_pid":1532108, "host_tid
       ":1535513,···}},
15
```

Listing 5.1 Example of log entries generated using *μProv*

a host. Listing 5.1 shows a sample log snippet from our logging framework. The log entries are enriched with four categories of log information: *Event Context*, *Task Context*, *Arguments*, and *Artifacts*. To track the system-level events, we configure kernel-space *Tracepoints* to hook syscall entry and exit events by executing *eBPF probes*. Another critical data structure is the *eBPF Map*. These maps are used to exchange data between the user space and the kernel space. We have used two eBPF maps: (i) arg_map that collects information from tracepoints; when the syscall entry probe is activated, the current process and thread IDs (PIDs and TIDs) are stored as the map index, with the syscall arguments saved as the corresponding values, and (ii) $exec\_map$ that stores the executable's absolute path during execution, essential for tracing log sources. When the syscall exit probe triggers, it uses the PID to retrieve process context from the arg_map and appends executable paths. Entries are added at the start of the process and removed upon termination to avoid performance issues. Additionally, a host-specific eBPF ring buffer is maintained to ensure causal ordering across microservices on a host. The logging container is deployed as a privileged container to have visibility for both the container and host PID namespace. To distinguish logs from different containers, our framework embeds a "*Task Context*" in ap-

plication logs (see listing 5.1). This is essential for separating log contexts across processes in various containers. In fig. 5.3, we illustrate how our framework performs atomic system event logging to produce causally consistent logs for microservices operating on a single host. The Log Collector collects log entries from multiple logging containers. To maintain causally consistent logging across hosts with differing clocks, our framework utilizes a "*Vector Clock*" [129], implemented in kernel space and triggered by each application event. The global log file is given input to the *Provenance Graph Generator* module to create the runtime provenance graph for the whole system.

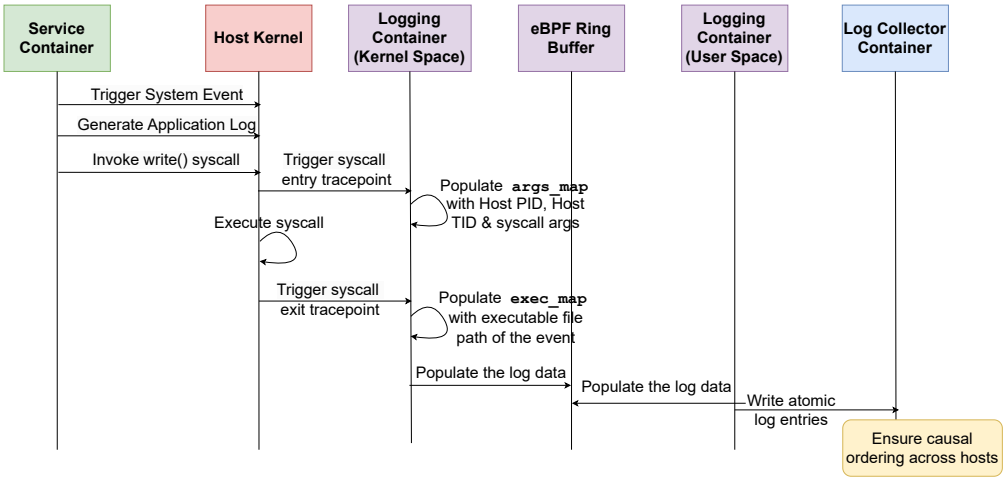


Figure 5.3 Atomic System Event Logging

5.2.2 Provenance Graph Generator

This component processes the runtime streaming global logs and converts them into a provenance graph for analysis. The *Provenance Graph Generator* uses the global file $\mathcal{L}$ to generate $\mathcal{G}$. A provenance graph $G = \langle V, E \rangle$ is a DAG that models the system log $\mathcal{L}$, maintaining a temporal and causal ordering of events. Each node $v \in V$ is an entity, which is either a system event or a resource (like open file descriptors), and the edges E capture the relationship among them. Notably, $\mathcal{L}$ contains interleaved application-level logs and Syslog entries. The Algorithm takes the log file and parses each log entry to either retrieve an existing node or create a new one. Each system call is processed individually, potentially adding new nodes and edges to the graph. After processing each log entry i, we maintain

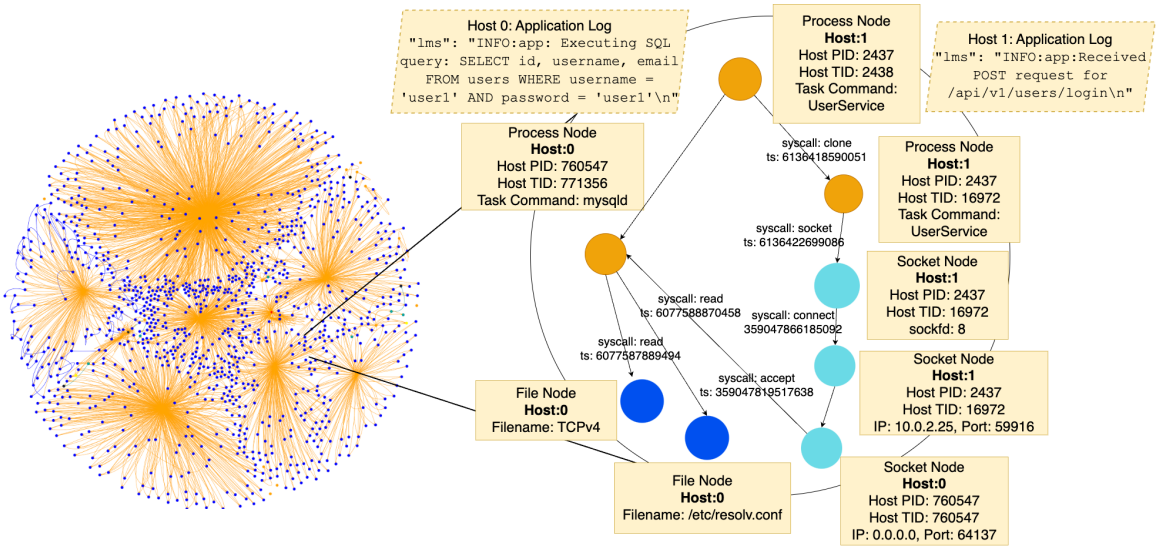


Figure 5.4 Provenance Graph generated from the global log file (Benign) for the *PicShare* application (fig. 5.6). The magnified section shows the causal path across multiple hosts when a new user registers.

and update a mapping $\mathcal{M}_i$ to maintain the relationships between various entities (e.g., processes, files, sockets). Each edge $e \in E_i$ includes a timestamp τ, preserving the temporal order of system calls.

For a given host system $\mathcal{H}$, let the whole-system log $\mathcal{L}$ be a continuously growing file that records every interaction within the operating system and its applications. Each entry $l_i = (n_i, n_j, s_k, t)$ in $\mathcal{L}$ represents a directed edge, where a system call s_k occurs between a source entity n_i and a destination entity n_j at a specific time t. The system entities belong to one of three types: process, file, or socket, and we focus on 21 distinct system calls[3] (1 *File I/O related:* `read`, `write`, `open`, `close`, `dup`, `dup2`, `dup3`, `openat`, `unlinkat`; 2 *Process related:* `clone`, `fork`, `vfork`, `execve`, `exit`, `exit_group`; and 3 *Socket related:* `connect`, `accept`, `bind`, `send`, `recv`, `socket`). Each pair of entities and system calls is unique; so for any two system calls (n_i, n_j, rel_p) and (n_j, n_i, rel_q), it holds that $rel_p \neq rel_q$, where rel is the relation between the two entities (*ex.*, a syscall *opens* a file descriptor). The direction of each system call intuitively represents the flow of information within the system, and each call uniquely defines a specific flow, ensuring no system call represents more than one flow.

[3]We've selected 21 syscalls for our implementation, but more can be added by defining entry and exit probe behavior in the logging framework.

Algorithm 3: Provenance Graph Generation (Part 1)

Procedure `Main`
  Input: $\mathcal{L}$: Global log file
  Output: $G = (V, E)$: Provenance graph
  $V \leftarrow \{v_{\text{exit}}, v_{\text{error}}\}$, $E \leftarrow \varnothing$;
  $\mathcal{M}_p, \mathcal{M}_f, \mathcal{M}_c, \mathcal{M}_n \leftarrow \varnothing$;
  foreach $l \in \mathbb{L}$ **do**
    $(h, p, t, \tau, s, c, m, pns) \leftarrow$ ExtractInfo(l);
    $v \leftarrow$ `CreateProcessNode`$(h, p, t, c, m, pns, \mathcal{M}_p)$;
    $(V', E', \mathcal{M}') \leftarrow$ `ProcessSyscall`$(s, v, G, \mathcal{M}, \tau, l)$;
    $G \leftarrow$ `UpdateGraph`(G, V', E');
    $(\mathcal{M}_p, \mathcal{M}_f, \mathcal{M}_c, \mathcal{M}_n) \leftarrow$ `UpdateMappings`$((\mathcal{M}_p, \mathcal{M}_f, \mathcal{M}_c, \mathcal{M}_n), \mathcal{M}')$;
  return G

Function `CreateProcessNode`$(h, p, t, c, m, pns, \mathcal{M}_p)$
  if $(h, p, t, c, m, pns) \notin \mathcal{M}_p$ **then**
    $v \leftarrow$ CreateNewProcessNode(h, p, t, c, m, pns);
    $\mathcal{M}_p[(h, p, t, c, m, pns)] \leftarrow v$;
  return $\mathcal{M}_p[(h, p, t, c, m, pns)]$

Function `ProcessSyscall`$(s, v, G, \mathcal{M}, \tau, l)$
  $V' \leftarrow \varnothing, E' \leftarrow \varnothing, \mathcal{M}' \leftarrow \varnothing$;
  switch s **do**
    case *accept4* **do**
      $(V', E', \mathcal{M}') \leftarrow$ ProcessAccept4$(v, l, G, \mathcal{M}, \tau)$;
    case *read* **do**
      $(V', E', \mathcal{M}') \leftarrow$ ProcessRead$(v, l, G, \mathcal{M}, \tau)$;
    case *openat* **do**
      $(V', E', \mathcal{M}') \leftarrow$ ProcessOpenAt$(v, l, G, \mathcal{M}, \tau)$;
  return $(V', E', \mathcal{M}')$

Algorithm 4: Provenance Graph Generation (Part 2)

Function `UpdateGraph`(G, V', E')
  $V \leftarrow V \cup V'$;
  $E \leftarrow E \cup E'$;
  return G

Function `UpdateMappings`$((\mathcal{M}_p, \mathcal{M}_f, \mathcal{M}_c, \mathcal{M}_n), \mathcal{M}')$
  foreach $(k, v) \in \mathcal{M}'$ **do**
    switch *type of* k **do**
      case *Process key* **do**
        $\mathcal{M}_p[k] \leftarrow v$;
      case *File descriptor key* **do**
        $\mathcal{M}_f[k] \leftarrow v$;
      case *Clone key* **do**
        $\mathcal{M}_c[k] \leftarrow v$;
      case *Network key* **do**
        $\mathcal{M}_n[k] \leftarrow v$;
  return $(\mathcal{M}_p, \mathcal{M}_f, \mathcal{M}_c, \mathcal{M}_n)$

Function *ProcessAccept4*$(v, l, G, \mathcal{M}, \tau)$
  $V' \leftarrow \varnothing, E' \leftarrow \varnothing, \mathcal{M}' \leftarrow \varnothing$;
  $fd \leftarrow l$.arguments.fd;
  $ip \leftarrow l$.artifacts.IP;
  $port \leftarrow l$.artifacts.$port$;
  if $(ip, port) \notin \mathcal{M}_n$ **then**
    $v_{net} \leftarrow$ CreateNetworkNode$(ip, port)$;
    $V' \leftarrow V' \cup \{v_{net}\}$;
    $\mathcal{M}'[(ip, port)] \leftarrow v_{net}$;
  $v_{net} \leftarrow \mathcal{M}_n[(ip, port)]$;
  $E' \leftarrow E' \cup \{(v, v_{net}, \text{"accept4"})\}$;
  $\mathcal{M}'[(v.pid, fd)] \leftarrow v_{net}$;
  return $(V', E', \mathcal{M}')$

Figure 5.5 Algorithm 3 and 4 jointly describe the procedure for generating the provenance graph from logs generated using our eBPF-based Logging Framework

Incremental Graph Construction: Algorithm 3 & 4 shows the runtime method for incremental provenance graph generation. Let $\mathcal{L} = \{l_1, l_2, ..., l_n\}$ be the sequence of log entries. We define the graph $G_i = (V_i, E_i)$ after processing the i-th log entry, and $\mathcal{M}_i$ as the set of mappings at step i. The following recurrence relations define the incremental construction:

$$V_i = V_{i-1} \cup V_i' \tag{5.1}$$

$$E_i = E_{i-1} \cup E_i' \tag{5.2}$$

$$\mathcal{M}_i = \text{UpdateMappings}(\mathcal{M}_{i-1}, \mathcal{M}_i') \tag{5.3}$$

where V_i', E_i', and $\mathcal{M}_i'$ are the new nodes, edges, and mapping updates produced by processing the i^{th} log entry. For each system call s, we define a function:

$$f_s : (V \times G_{i-1} \times \mathcal{M}_{i-1} \times \tau) \rightarrow (V_i' \times E_i' \times \mathcal{M}_i') \tag{5.4}$$

The fig. 5.4 shows a sample output of the runtime *Provenance Graph Generation* Algorithm. The magnified area shows the causal path for when a new user registers; it goes through the user service running on Host-1 and the information (username and password) stored in the MySQL database running on Host 0). The rhombus shows the corresponding application logs for generating the system call path. The runtime attack detection module can extract the features from this graph to train ML models to detect possible security attacks, as we discussed next with a PoC application.

5.3 Proof-of-concept Implementation

As discussed earlier, a key challenge in attack detection research is that most open-source benchmarking lacks the complexity to simulate multi-stage attacks that exploit vulnerabilities commonly identified by CVEs and CWEs. Therefore, we develop a microservice-based application named *PicShare* (refer fig. 5.6) to emulate real-world vulnerabilities for distributed microservices.

5.3.1 *PicShare* Application: An Attack Emulation Platform

The application implements an end-to-end service for uploading pictures, as well as getting recommendations and notifications on the activity of the pictures. *PicShare* is a dockerized microservice application and supports Docker Swarm for running in multiple hosts. We emulate various attacks on this application, as summarized in table 5.1.

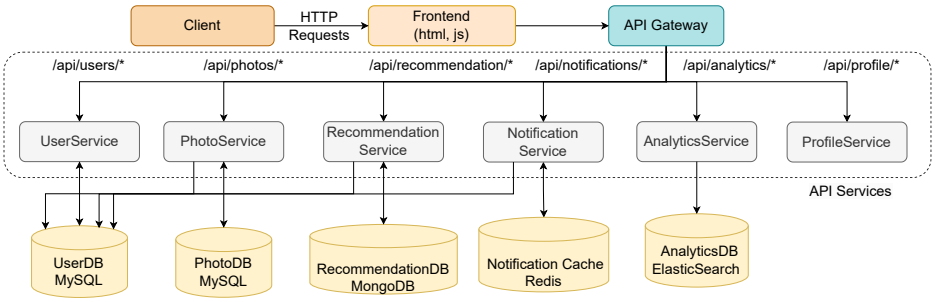


Figure 5.6 The architecture of the `PicShare` Service for uploading, sharing, and viewing photos.

Functionalities: In the application, users interface with a *node.js* front-end to register and log into their account. Once logged in, they can upload a photo from their user end and view a particular user's photos. The microservices are written in Python Flask, while the back-end databases consist of a relational database MySQL, persistent MongoDB instances, and a Redis DB. It contains the following microservices: (1) a front-end server implemented using *NodeJS-EJS Templating* that serves users' requested HTML pages. It handles the application's presentation layer, facilitating dynamic content rendering. (2) *PhotoService* implemented using Python Flask to upload and view photos. This component handles the back-end processing necessary for uploading images securely and efficiently. (3) *UserService* to register and log in to the account. (4) *RecommendationService* An open-source unified analytics engine utilized for the recommendation engine of *PicShare*. (5) A *NotificationService* (using Flask) enables interactions related to likes, dislikes, and other user preferences. It provides a streamlined interface for communication with the recommendation engine. (7) *ProfileService* implemented using NodeJS to view the user profile. (8) *AnalyticsService* (implemented using ElasticSeach): A visualization tool integrated with ElasticDB, employed by application developers to visualize the performance metrics of the component. This application exposes 7 endpoints (*downloadphotos, getprofile, getrecom-*

mendation, loginuser, registeruser, updaterecommendation, viewphoto) to users labeled as 'benign' $\{L_1, L_2, L_3, L_4, L_5, L_6, L_7\}$.

Attack Execution: In the context of microservices-based environments, applications are divided into smaller services, each potentially vulnerable to various types of attacks: (1) *Communication/Network Attacks*, such as Denial of Service, Man-in-the-Middle, Replay, etc., (2) *Service Level Attacks*, such as Injection (such as SQL injection and Shell Injection), Broken Access Control, Cryptographic Failures, Server-Side Request Forgery (SSRF), etc., (3) *Virtualization Attacks*, such as RCE in the host, Unauthorized Access, Container Malware, etc. We have emulated a subset of these attacks on the *PicShare* applications, as shown in table 5.1, each identified with its corresponding CVE and CWE identifiers and categorized attack types labeled as $\{L_8, L_9, L_{10}, L_{11}, L_{12}\}$. These vulnerabilities are selected based on their compatibility with the targeted versions and components of the system.

Table 5.1 Injected Vulnerabilities for Attack Emulation

Label	Attack Scenario ID	Attack Type	Vulnerability Description	Affected Microservices	Expected Log Patterns
L_8	SSTI-01 CVE-2024-29686	Server-side Template Injection	Remote attacker to execute arbitrary code via a crafted payload	ProfileService	Received request for username: <%= 7 * 7 %>
L_9	PATH-TRAV-001 CVE-2019-5418	Path Traversal	Improper input sanitization allows access to files outside the intended directory and remote code execution	PhotoService	Access attempts to sensitive files, unexpected file access errors
L_{10},L_{11}	SQL-INJ-001 CVE-2014-3704	SQL Injection	SQL queries are constructed using string interpolation, allowing user input to manipulate the query structure	UserService PhotoService	SQL syntax errors, unexpected query results,
L_{12}	INS-FILE-001 CVE-2020-36112	Insecure File Upload	Improper validation of uploaded files allows remote code execution	PhotoService	Unusual file types, file handling errors

5.3.2 Attack Detection Techniques

User requests exhibit diverse behaviors that trigger distinct system call sequence signatures during execution. Our objective is to analyze these interaction patterns and signatures to differentiate between benign and malicious requests as well as between different types of malicious requests. Given that user-facing applications often include multiple operations (or endpoints), we treat each request type as a separate class, resulting in 12 different types. This allows us to approach attack detection as a multiclass classification problem with 7 benign and 5 malicious request classes.

(i) Feature Extraction: We extract relevant features from both log data $\mathcal{L}$ and the provenance graphs $\mathcal{G}$. Initially, we construct a provenance graph from the global log file for each request, resulting in a set of graphs G_i, where each graph is labeled according to its respective class as either benign or attack, with its specific type $L_i \in \{L_1, L_2, \ldots, L_{12}\}$ (refer to section 5.3). From each graph G_i, we derive basic graph-related features, such as the number of nodes, edges, average degree, density, degree centrality, number of connected components, node connectivity, edge connectivity, in-degree and out-degree centrality, and degree associativity. Additionally, we extract specialized features from the provenance graphs, including the number of system calls, system call frequencies, system call counts, and their return values. While graph-related features help us efficiently detect malicious requests, provenance-specific features allow us to identify subtle changes in system call sequences that could indicate small-scale attacks, which might otherwise resemble benign requests.

(ii) Model Selection: We explore standard supervised multiclass classification techniques commonly employed in traditional frameworks for detecting various benign and attack scenarios [47, 50]. Specifically, we investigate classification models such as K-Nearest Neighbors (KNN) [141], Support Vector Machines (SVM) [88, 141], Random Forest [12], and Artificial Neural Networks (ANN) [142], as these models have been shown to perform well with high-dimensional numerical features. We implement versions of these ML models for the multiclass classification task, training each model with labeled features extracted from the graphs. All models were trained on the same set of features and their vectorized representations. Specifically, for each graph G_i along with the corresponding label L_i, we extract all relevant features and represent them as a one-dimensional vector, where the length of the vector corresponds to the total number of features extracted.

These feature vectors are then used as input for training and testing the attack detection models.

Table 5.2 Performance Comparison for our Framework

Model Name	**Tracee**				***μProv***			
	Accuracy	Precision	Recall	Micro-F1 Score	Accuracy	Precision	Recall	Micro-F1 Score
KNN	48.27	47.76	46.66	46.18	70.36	70.39	71.21	70.61
SVM	38.69	42.23	40.18	39.36	83.14	81.14	80.25	80.12
Random Forest	52.80	54.93	52.80	53.10	87.78	87.97	87.78	87.64
ANN	46.32	56.54	48.81	48.72	83.05	89.90	90.18	88.21

5.4 Evaluation

We evaluate *μProv* with the following objectives: (1) the accuracy in correctly identifying the attack scenarios in comparison to Tracee[4], a widely used system auditing framework that captures runtime syscall execution logs using eBPF, and analyzing various scenarios in terms of performance-accuracy trade-off, and (2) the runtime resource consumption of *μProv* in comparison to Tracee.

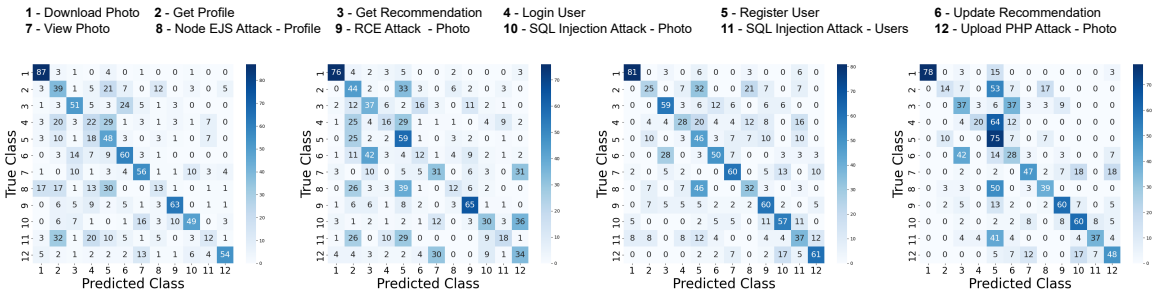
5.4.1 Experimental Setup

The source code, documentation, and experimental configuration scripts for our implementation have been open-sourced[5]. We used C with the *libbpf* library to create eBPF probe programs, identifying 21 configurable syscalls for which we developed separate eBPF probes. For the kernel-space component, we set the eBPF ring buffer size to 1MB with a 50ms polling interval, forwarding logs to the Logging collector. Additionally, we reserved 2MB for the $exec\_map$ and 64KB for the $args\_map$. Provenance Graph Generator is a Python program with approx 880 lines of code.

To assess the effectiveness of our framework, we employed a vulnerable photo-sharing microservice application, *PicShare*, consisting of 13 distinct microservices. The main user interaction endpoints include the *UserService (US)*, *PhotoService (PS)*, *Recommendation-Service (RS)*, and *ProfileService (PS)*. We developed an automation script that generates 1000 requests, spaced 5 seconds apart, targeting both benign and attack scenarios across services with known vulnerabilities (refer to table 5.1). In total, we gathered logs for 12 scenarios (7 benign and 5 attack scenarios as labelled as L_1 to L_{12}). To evaluate the framework's ability to collect logs across multiple hosts, we deployed the microservices on 10 virtual machines (VMs) configured as edge computing hosts. Each VM hosted several containerized microservices, managed by Docker Swarm, and was equipped with Ubuntu 22.04 SMP, the Linux 6.5.0-41-generic kernel, 16 vCPU cores, and 64GB of RAM. One VM serve as the Docker Swarm manager, while the remaining VMs operated as worker

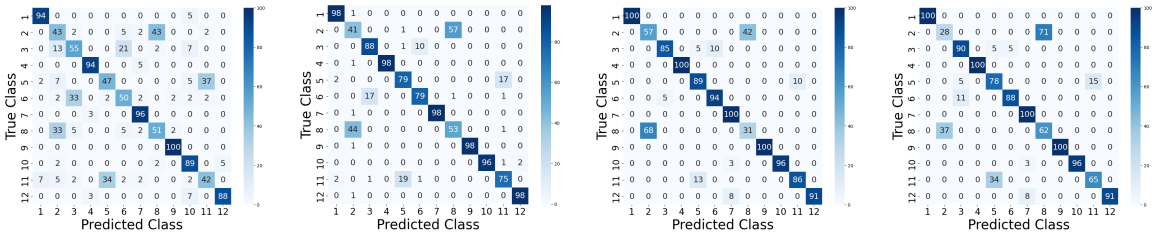
[4] https://github.com/aquasecurity/tracee (Accessed: May 5, 2026)
[5] https://github.com/usatpath01/MuProv (Accessed: May 5, 2026)

nodes.



(a) Tracee - KNN **(b)** Tracee - SVM **(c)** Tracee - Random Forest **(d)** Tracee - ANN

Figure 5.7 Confusion Matrix for Tracee (%)



(a) *µProv* - KNN **(b)** *µProv* - SVM **(c)** *µProv* - Random Forest **(d)** *µProv* - ANN

Figure 5.8 Confusion Matrix for *µProv* (%)

5.4.2 Results

(i) Attack Detection Performance

We have compared the performance metrics (Accuracy, Precision, Recall, and Micro-F1 Score) of four machine learning models (KNN, SVM, Random Forest, and ANN) for detecting attacks using Tracee and *µProv* as shown in table 5.2, Random Forest shows the highest performance in *µProv*, achieving an accuracy of 87.78%, a precision of 87.97%, and a Micro-F1 Score of 87.64%. In comparison, its performance with Tracee is significantly lower, with an accuracy of 52.80%. These results indicate that *µProv* consistently outperforms Tracee in every metric due to its causally-ordered comprehensive logging information.

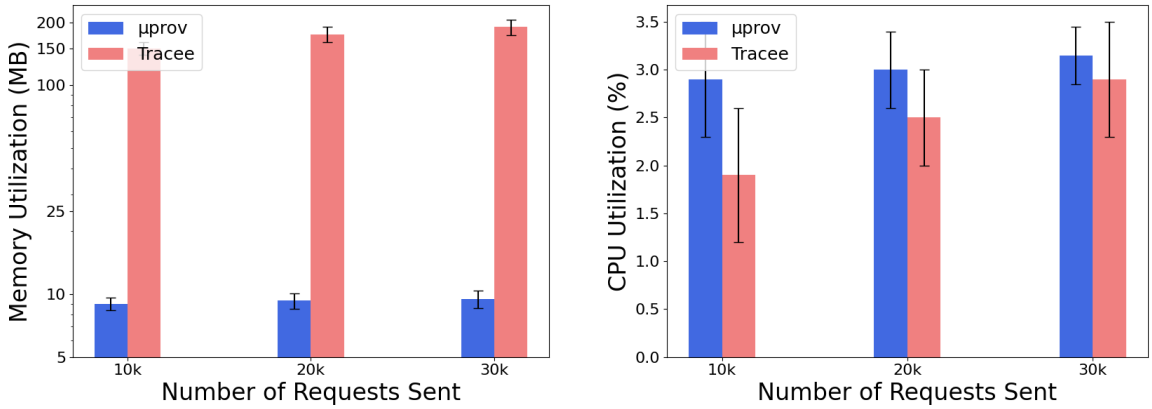
Figure 5.7 and fig. 5.8 show the confusion matrix of different machine learning models (KNN, SVM, Random Forest, and ANN) using Tracee and *μProv* in identifying attack scenarios across 12 classes of 'benign' and 'attack' activities. Across all the models, *μProv* consistently outperforms Tracee in classifying attack and benign events. This highlights the effectiveness of *μProv*'s causally-ordered logs, which provide a richer context for distinguishing. Also, it shows Random Forest exhibits the best overall performance for Tracee and *μProv*, as demonstrated by the higher diagonal values in the matrix. Tracee fails to classify attacks like SQL Injection and RCE, with confusion spread across neighbors. This is due to the similarity in the attack signature and overlapping patterns in the system call sequences between these events, particularly when they are evaluated on individual hosts in silos. In comparison, the causality-aware logging provided by *μProv* provides a robust whole-system provenance with better classification accuracy for these scenarios.

We also measure each model's False Positive (FP) and False Negative (FN) using fig. 5.7 and fig. 5.8. FP is when a benign event is misclassified as an attack, or one type of attack is misclassified as another. FN is when an attack is classified as benign activity or a different attack. Tracee has higher rates of FP and FN. As shown in (a), the KNN model in Tracee shows significant FP rates for benign activity classified as an attack. In contrast, *μProv* improves FP rates across all models with fewer benign activities misclassified as attacks. Also, Tracee exhibits higher FN rates across all models. For example, the *SQL injection Attack (User)* is often misclassified as benign activities *Get Profile* or *View Photo* when the SVM model is used over Tracee-generated logs. On close inspection, we observe that the syscall-execution sequences for these activities are similar when observed on individual hosts. As Tracee fails to provide a holistic view of the entire system, such differences are not observed when the provenance graph is constructed using Tracee-generated logs. In contrast, *μProv* reduces the number of FP (benign activity falsely detected as attacks) and FN (missed attacks) by combining the correlated logs from individual hosts, leading to better overall classification accuracy for attack detection.

We compare the attack detection time between *μProv* and Tracee. For Tracee, the process includes generating logs across hosts, pre-processing, storing them centrally, generating a provenance graph, and running the pre-trained detection model. *μProv*, however, streams logs directly to a central server, eliminating the need for relevant log retrieval. We observe that on average, *μProv* detects attacks in 5.2 ± 0.5 seconds while Tracee takes

10.2 ± 0.6 seconds from the attack occurrence, making *μProv* ~ 50% faster.

While causal ordering is a major contributor to the observed improvement in detection accuracy, it is not the sole factor influencing performance. The improved results of *μProv* can be traced back to a number of architectural benefits, such as cross-host provenance reconstruction, which gives a more complete view of execution than per-machine methods can. Additionally, the more detailed context data gathered through eBPF instrumentation allows for the creation of better provenance graphs, improving the ability of the downstream ML model to distinguish between different cases. To ensure fairness, the downstream feature extraction and machine learning detection pipeline remain conceptually consistent across both implementations. The primary difference lies in the log collection and causal correlation architecture, which influences the quality of the resulting provenance graph.



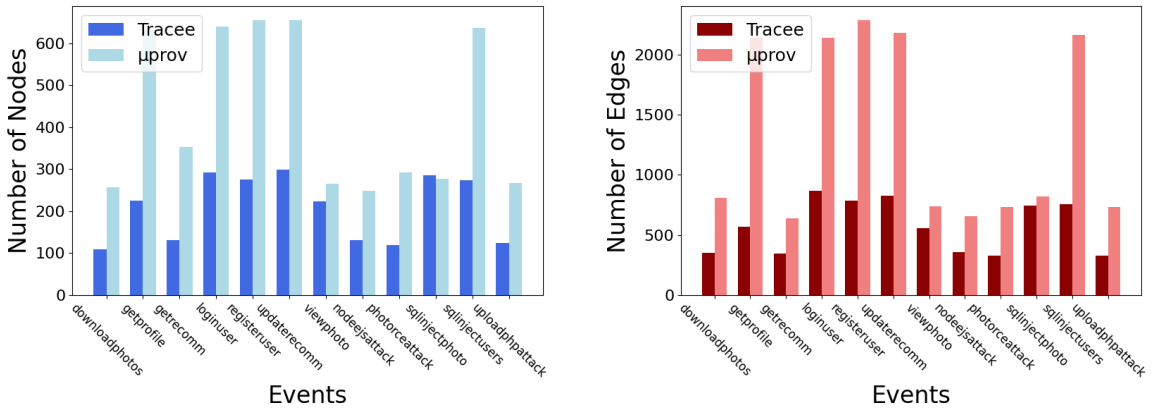
(a) Avg. Memory Usage **(b)** Avg. CPU Utilization

Figure 5.9 Average memory overhead and CPU utilization

(ii) Resource Utilization

To evaluate the resource overhead, we have used an open-source monitoring tool developed by Google to monitor containers, *cAdvisor*[6]. As shown in fig. 5.9a, the memory overhead of the logging framework of *μProv* remains stable across 10K, 20K, and 30K requests, around 10MB, whereas Tracee takes around 200MB. Figure 5.9b shows the CPU utilization overhead by *μProv* and Tracee. The results shown in fig. 5.10 describe the properties

[6] https://github.com/google/cadvisor (Accessed: May 5, 2026)



(a) No of Nodes Generated per Event **(b)** No of Edges Generated per Event

Figure 5.10 Properties of Provenance Graph

of the generated provenance graph for *µProv* and Tracee across 12 different events. As shown, *µProv* consistently generates more nodes and edges than Tracee across all events (although with a much lower memory footprint than Tracee), indicating a more detailed and comprehensive provenance tracking approach with a lower resource consumption footprint.

5.5 Summary

In this chapter, we developed a lightweight and application-agnostic provenance-based framework named *µProv* for accurate and real-time attack investigation over distributed microservice-based applications. *µProv* is designed to capture fine-grained system interactions across microservices by generating causally ordered, dynamic provenance graphs without relying on instrumentation of the application. At its core, *µProv* leverages eBPF to collect both system and application-level events from multiple hosts, ensuring causal consistency through vector clocks and ring buffers. These enriched logs are then incrementally converted into provenance graphs that capture the causal flow of system calls, file access, and network interactions. To address the lack of reproducible attack datasets, we developed PicShare, a vulnerable microservice application instrumented with known CVEs and CWEs to emulate complex, real-world multi-stage attacks. Using PicShare, we generated 12 scenarios (7 benign and 5 malicious) and built a labeled dataset of provenance graphs. We extracted both graph-structural and provenance-specific features and trained traditional

machine learning models like Random Forest and SVM for multiclass attack detection. Our empirical analysis showed that *μProv* consistently outperformed existing logging solutions like Tracee across accuracy, detection time, and resource utilization. For instance, Random Forest achieved 87.78% accuracy and reduced attack detection time by nearly 50%. *μProv*'s causally ordered logs also helped reduce false positives and false negatives compared to siloed host-based logs. Our evaluation demonstrates that *μProv* is practical for deployment in real-world containerized microservices to generate detailed, dynamic provenance graphs and perform scalable attack detection. In the next chapter, we extend this work to incorporate advanced graph learning techniques over the generated provenance graphs for unsupervised anomaly detection in evolving microservice environments.

Chapter 6

Attack Detection in Microservices via Enriched Provenance Graphs and GAEs

Distributed microservice applications are increasingly adopted in modern computing due to their scalability, flexibility, and ease of deployment [134, 135], offering clear advantages over traditional monolithic architectures. However, deploying these applications as containers across multiple physical servers introduces complex inter-service communication, complicating the task of detecting anomalous or malicious behavior [11, 136–138]. Although containerization improves deployment agility, it also reduces isolation: containers share the host operating system's kernel, making them susceptible to threats such as container escape and privilege escalation attacks, typically executed through malicious system calls.

As systems grow more distributed, achieving secure and reliable operations requires comprehensive observability, capturing and analyzing system events across services and hosts. Observability not only involves detecting system anomalies but also understanding inter-service dependencies and behavior, which is crucial for root-cause analysis. In microservice-based deployments, a single request can trigger a chain of interactions spanning multiple hosts. However, existing provenance tracking mechanisms often treat each host individually [12, 43, 46, 50], making it difficult to correlate low-level system logs with high-level application behavior in a distributed setting. Provenance graphs are commonly

used to model such interactions, representing entities (e.g., processes, files, sockets) as nodes and their interactions as edges.

Advanced persistent threats (APTs) often span both the application and infrastructure layers. Accurately detecting such threats requires correlating events from disparate layers, including virtual machines, containers, and service-level sandboxes. These layers frequently implement different namespace semantics; for instance, containers use separate PID namespaces, which may result in identical PIDs representing different entities across layers. This makes constructing a consistent and accurate runtime provenance graph particularly challenging.

Despite progress in graph-based anomaly detection, key limitations remain. **(1)** Current provenance graphs rely on coarse-grained representations that do not fully exploit the rich, contextual data made available by modern logging tools. Attributes such as file access modes, socket IP addresses, and port information are often ignored, limiting the expressive power of these graphs and hindering the performance of downstream learning models. **(2)** Most state-of-the-art approaches employ supervised learning techniques [49–51, 107], which depend on labeled anomaly data. However, in real-world deployments, labeled datasets are rare and often fail to capture evolving or zero-day attacks, resulting in poor generalizability and robustness.

To address these gaps, we propose ***μProvGAE***, an unsupervised anomaly detection framework built on enriched heterogeneous provenance graphs. This work extends our previous system, ***μProv***, by incorporating fine-grained logging and advanced graph learning techniques to improve detection in distributed microservice environments. Our key contributions are:

• Contribution (1): *Rich, Heterogeneous Provenance Graph Construction.* – We enhance the eBPF-based logging framework of *μProv* to construct a more expressive provenance graph that captures both causal interactions and detailed node-specific attributes. These include file names, access flags, modes, socket IPs, and ports, enabling more comprehensive behavioral modeling across distributed microservices.

• Contribution (2): *Tailored Heterogeneous Graph Autoencoder (GAE).* – We design a cus-

tom heterogeneous GAE capable of handling diverse node and edge types, along with their associated attributes. This encoder maps both structural and semantic properties of the graph into a unified latent space, enabling the system to learn meaningful representations of normal system behavior.

• Contribution (3): *Unsupervised Graph-level Anomaly Detection.* – Our method identifies anomalous request patterns by comparing runtime provenance graphs to learned normal patterns, entirely without relying on labeled attack data. This unsupervised approach ensures adaptability to unknown or evolving threats, making it suitable for real-world deployment.

• Contribution (4): *Unsupervised Graph-level Anomaly Detection.* – Empirical Evaluation with Containerized Application: We evaluate *μProvGAE* on a realistic containerized photo-sharing application under real-world attack scenarios. Our system achieves higher detection accuracy and a lower false positive rate compared to traditional ML baselines, validating its effectiveness and practical relevance.

6.1 Design Goals & System Overview

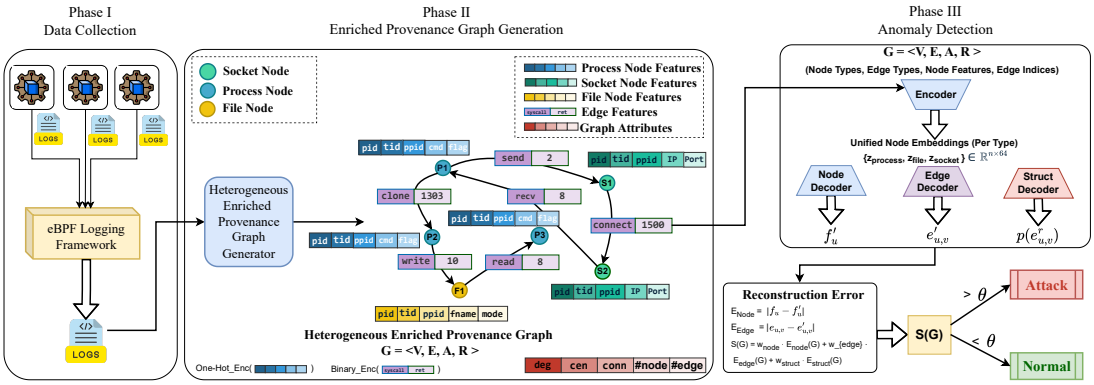


Figure 6.1 *μProvGAE*'s Architecture

As mentioned, *μProvGAE* builds on the foundation of our previous work *μProv*. We introduce an enhanced architecture that combines enriched heterogeneous provenance graphs with unsupervised learning for anomaly detection, which is created from the causally-

consistent log message streams collected using a distributed log collection architecture from running microservices over the sandboxed distributed platforms. In this section, we first formally define the concept of causally-consistent log message stream, heterogeneous enriched provenance graph and then discuss the broad overview of *μProvGAE*.

Component 1 (Heterogeneous Enriched Provenance Graph): An heterogeneous provenance graph G = $\langle V, E, A, R \rangle$ where V = $V_1, V_2, ..., V_n$ is the set of nodes classified into n types where n∈{process, file, socket}; E = $\{E_1, E_2, ..., E_m\}$ is the set the edges classified into m types where m ∈ {read, write, open, close, connect, accept, etc.}; A = $A_1, A_2, ..., A_n$ is the set of node feature vectors, where $A_i \in \mathbb{R}^{(|V_i| \times d_i)}$ and d_i is the feature dimension for node type n; R = $\{R_1, R_2, ..., R_m\}$ is the set of edge feature vectors, where $R_j \in \mathbb{R}^{(|E_j| \times f_j)}$ and f_j is the feature dimension for edge type m.

Component 2 (Unsupervised Anomaly Detection): Given the set of enriched provenance graphs $G = \langle G_1, G_2, ...G_k \rangle$ which represents the normal system behaviour, our objective is to learn a function $f : G \rightarrow Z$ that maps each graph G to a latent representation Z. Also, to learn a reconstruction function $g : Z \rightarrow G'$ that reconstructs the original graph from its latent representation. An anomaly score is defined as $S(G)$ based on the reconstruction error between G and G'. If $S(G_{new}) > \theta \Rightarrow G_{new}$ is anomalous, where G_{new} is a new provenance graph and θ is a threshold determined from the distribution of reconstruction error on the normal graph.

Component 3 (Graph-level Anomaly Score): The graph-level anomaly score $S(G)$ is defined as:

$$S(G) = \alpha \cdot L_{\text{node}}(G, G') + \beta \cdot L_{\text{edge}}(G, G') + \gamma \cdot L_{\text{struct}}(G, G') \tag{6.1}$$

Where $L_{node}(G, G'), L_{edge}(G, G')$, and $L_{struct}(G, G')$ calculate the node feature reconstruction error, edge reconstruction error and graph structure reconstruction error, respectively (Refer Equation 10, 11, 12). Node feature reconstruction error is calculated as the Mean Square Error (MSE) between the original and reconstructed node feature vector. The edge reconstruction error compares the predicted adjacency matrix to the original one using a binary cross-entropy or similar loss function. The graph structural reconstruction error captures the high-level topological inconsistencies between G and G' like subgraph pattern,

difference in degree distributions, number of components etc,. α, β, and γ are the hyperparameters used to control the relative importance of each component. The major challenge is to design an autoencoder architecture that can handle the heterogeneous nature of the provenance graph, learn meaningful latent representations from the normal behaviour pattern and produce the reconstruction error to differentiate between normal and anomalous graphs effectively.

We next discuss the broad overview of our proposed attack detection framework *μProvGAE*. As illustrated in fig. 6.1, *μProvGAE* works in three phases to achieve the above design goals: (I) the *Data Collection*, (II) the *Graph Generation and Representation*, (III) the *Anomaly Detection Module* that analyzes the graphs in real-time to detect potential attacks. In Phase I, we have a Logging framework deployed on each host, responsible for ensuring the causal consistency of log messages generated by microservices and aggregating these logs across hosts while preserving causal relationships; In Phase II, we have a Graph Generator module to dynamically constructs enriched heterogeneous provenance graphs from the collected logs and in Phase III, the Anomaly Detection Module that analyzes the graphs in real-time to detect potential attacks.

6.2 $\mu ProvGAE$'s Components

The architecture of *μProvGAE* (illustrated in fig. 6.1) comprises of 3 core components: (1) **eBPF-based Logging Framework** – captures fine-grained, causally-consistent logs from distributed microservices, (2) **Enriched Provenance Graph Generator** – constructs a heterogeneous provenance graph representing event causality and extracts corresponding node and feature features, and (3) **Anomaly Detection Module** – employs a heterogeneous graph autoencoders to learn the latent representation of normal system behaviour and detect anomalies based on reconstruction error. We describe each component in detail below.

6.2.1 eBPF-based Logging Framework

We design a logging framework leveraging eBPF [139] to collect both system-level and application-level logs across distributed microservices and produce a unified, causally ordered log $\mathcal{L}$. The framework is composed of three key stages: log collection, preprocessing, and causal ordering.

By embedding eBPF probes at kernel-level *tracepoints*, we enable efficient, application-independent logging without modifying kernel source code or loading additional modules. As shown in fig. 6.2, the framework includes: **(1) A Logging Container** deployed on each host, which ensures intra-host causal consistency of logs generated by local microservices, and **(2) A Log Collector Container** that aggregates logs from all hosts and maintains inter-host causality. These causally consistent logs are then streamed to the *Enriched Provenance Graph Generator*, which incrementally constructs the provenance graph as described in algorithm 5.

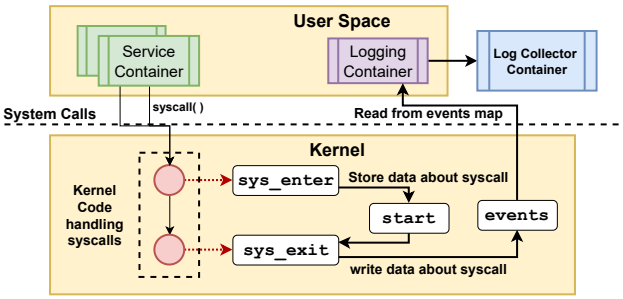


Figure 6.2 eBPF-based Logging Framework

The proposed logging framework monitors the system-level events using *eBPF* probes on system call entry and exit tracepoints, thus ensuring that all the relevant logs corresponding to a syscall event are generated simultaneously, therefore preserving the causal context for a microservice running over a host. Listing 6.1 shows a sample log snippet from our logging framework. The log entries are enriched with four categories of log information: *Event Context*, *Task Context*, *Arguments*, and *Artifacts*. To track the system-level events, we configure kernel-space *Tracepoints* to hook syscall entry and exit events by executing *eBPF probes*. Another critical data structure is the *eBPF Map*. These maps are used to exchange data between the user space and the kernel space. We have used two eBPF maps: (i) arg_map that collects information from tracepoints; when the syscall entry probe is activated, the current process and thread IDs (PIDs and TIDs) are stored as the map index,

with the syscall arguments saved as the corresponding values, and (ii) $exec\_map$ that stores the executable's absolute path during execution, essential for tracing log sources. When the syscall exit probe triggers, it uses the PID to retrieve process context from the arg_map and appends executable paths. Entries are added at the start of the process and removed upon termination to avoid performance issues. Additionally, a host-specific eBPF ring buffer is maintained to ensure causal ordering across microservices on a host. The logging container is deployed as a privileged container to have visibility for both the container and host PID namespace. To distinguish logs from different containers, our framework embeds a "*Task Context*" in application logs (see listing 6.1). This is essential for separating log contexts across processes in various containers.

In fig. 6.3, we illustrate how our framework performs atomic system event logging to produce causally consistent logs for microservices operating on a single host. The Log Collector collects log entries from multiple logging containers. To maintain causally consistent logging across hosts with differing clocks, our framework utilizes a "*Vector Clock*" [129], implemented in kernel space and triggered by each application event. The global log file is given input to the *Provenance Graph Generator* module to create the runtime provenance graph for the whole system.

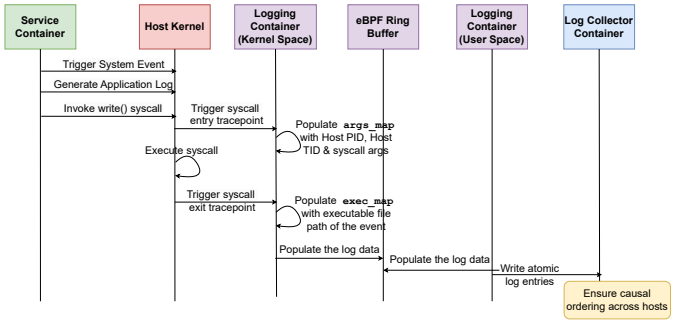


Figure 6.3 Atomic system event logging showing log flow from syscall entry to global collector

6.2.2 Enriched Provenance Graph Generator

The second core component is the provenance graph generator, which dynamically transforms the runtime streaming global logs and converts them into an enriched provenance graph $G = \langle V, E, A, R \rangle$ for anomaly detection. Here V represents the set of distinct node types, E represents the set of directed edges capturing the causal interactions, A comprises the node feature vector and the R contain the edge feature vector.

```
/* Application Log */
Host 1 : {"event_context": {"ts": 173989395151310,"datetime": "09:00:51","
    task_context": {"host_pid": 314463, "host_tid": 317863,"task_command" : "
    PhotoService" ··· }},
    "data": {"lms" : "Photo Service Response: {\n  filename: '1762071b-9466-48ef
    -87ae-cd956b34c4fb.png',\n  message: 'File uploaded successfully',\n  status:
     'success'\n}\n"},
    "artifacts":{"exe":"/usr/local/bin/PhotoService"}}

Host 2 : { ··· "task_context" : {"host_pid" : 1532108,"host_tid":1535513, ···}, "data"
     : { "lms" : INFO:app:User user99 registered successfully\n }, "artifacts":{"
    exe":"/usr/local/bin/UserService"}},
/* 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":"0x80003","addrlen":16},
    "artifacts": {"exe": "/usr/local/bin/PhotoService,"IP":"10.0.2.22","port
    ":"34835"}}
Host 1 : { ··· "syscall_name" : "read", "task_context" : { "host_pid" :  314463, "
    host_tid" :  317863, ··· "artifacts" : { "exe" : "/usr/local/bin/UserService",
     "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 6.1 Example of log entries generated using *µProv*

For a given host system $\mathcal{H}$, let the whole-system log $\mathcal{L}$ be a continuously growing file that records every interaction within the operating system and its applications. Each entry $l_i = (n_i, n_j, s_k, t)$ in $\mathcal{L}$ represents a directed edge, where a system call s_k occurs between a source entity n_i and a destination entity n_j at a specific time t. The system entities belong to one of three types: process, file, or socket, and we focus on 21 distinct system calls (refer table 6.1).[1]

Each pair of entities and system calls is unique; so for any two system calls (n_i, n_j, rel_p) and (n_j, n_i, rel_q), it holds that $rel_p \neq rel_q$, where rel is the relation between the two entities (*e.g.*, a syscall *opens* a file descriptor). The direction of each system call intuitively represents the flow of information within the system, and each call uniquely defines a specific flow, ensuring no system call represents more than one flow. table 6.1 shows the properties of the enriched provenance graph.

algorithm 5 shows the runtime method for incremental provenance graph generation.

[1]We've selected 21 syscalls for our implementation, but more can be added by defining entry and exit probe behavior in the logging framework.

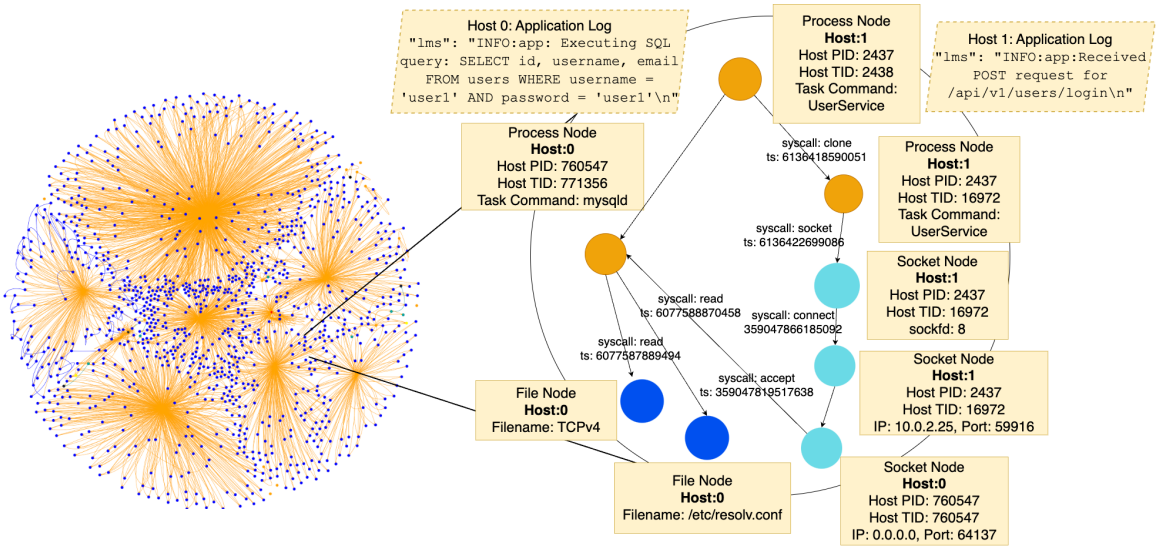


Figure 6.4 Provenance Graph generated from the global log file (Benign) for the *PicShare* application (fig. 6.5). The magnified section shows the causal path across multiple hosts when a new user registers.

Let $\mathcal{L} = \{l_1, l_2, ..., l_n\}$ be the sequence of log entries. We define the graph $G_i = (V_i, E_i, A_i, R_i)$ after processing the i-th log entry, and $\mathcal{M}_i$ as the set of mappings at step i. The following recurrence relations define the incremental construction:

$$V_i = V_{i-1} \cup V_i' \tag{6.2}$$

$$E_i = E_{i-1} \cup E_i' \tag{6.3}$$

$$\mathcal{M}_i = \text{UpdateMappings}(\mathcal{M}_{i-1}, \mathcal{M}_i') \tag{6.4}$$

where V_i', E_i', and $\mathcal{M}_i'$ are the new nodes, edges, and mapping updates produced by processing the i^{th} log entry. For each system call s, we define a function:

$$f_s : (V \times G_{i-1} \times \mathcal{M}_{i-1} \times \tau) \rightarrow (V_i' \times E_i' \times \mathcal{M}_i') \tag{6.5}$$

fig. 6.4 shows a sample output of the Phase I of runtime *Enriched Provenance Graph Generation* Algorithm. The magnified area shows the causal path for when a new user registers; it goes through the user service running on Host-1 and the information (username and password) stored in the MySQL database running on Host 0). The rhombus shows the corresponding application logs for generating the system call path.

Table 6.1 Enriched Provenance Graph Properties

Graph Component	**Example Type**	**Node Features ⟨ task_context, args, artifacts ⟩**
Node	Process (P)	⟨ host_pid, host_tid, host_ppid, pid, tid, ppid, cgroup_id, mntns_id, pidns_id, task_cmd flags, parent_tid, child_tid, tls, cmdline args ⟩
	File (F)	⟨ host_pid, host_tid, host_ppid, pid, tid, ppid, cgroup_id, mntns_id, pidns_id, task_cmd filename, flags, mode, exe, epoch ⟩
	Socket (S)	⟨ host_pid, host_tid, host_ppid, pid, tid, ppid, cgroup_id, mntns_id, pidns_id, task_cmd fd, uservaddr, addrlen, epoch, IP, port, exe, epoc ⟩
Edge	**Example Type**	**Edge Features ⟨ syscall_name, return_code ⟩**
	P → P	⟨ clone, 2 ⟩, ⟨ fork, 35 ⟩, ⟨ execve, 10⟩
	P → F	⟨ read, 20⟩, ⟨ write, 45⟩, ⟨ open, 64⟩, ⟨ close,32⟩
	F → P	⟨ read, 24⟩, ⟨ execve, 67⟩,
	F → F	⟨ dup, 10⟩, ⟨ linkat, 15⟩, ⟨ unlinkat, 8⟩
	P → S	⟨ socket, 28⟩, ⟨ bind, 45⟩, ⟨ connect, 64⟩, ⟨ send,32>
	S → P	⟨ accept, 15⟩, ⟨ recv, 65⟩
	S → S	⟨ connect, 64⟩, ⟨ send,32⟩, ⟨ recv, 65⟩

Once the provenance graph is generated then we create the node feature matrix for all nodes and the edge index and edge attributes to capture various attributes of the same.

(i) Node Feature Extraction: The node feature vector dimension is 162, which consists of Node type dimension (number of node types), HID dimension (8), PID dimension (16), TID (16), Commandline (32), filename (64), and IP/Port dimension (16). The total node feature is the sum of all these dimensions in addition to the node type dimension ($\mathcal{T}$), which varies based on how many different types of nodes are in the system. The node features capture information about the process, files and network connections, while the edge features capture the information about the system calls that connect these nodes. These dimensions are chosen to balance the information capacity, i.e. bits to represent the data without context loss, memory efficiency, i.e. no much memory wastage and practical requirements like system PID ranges.

(ii) Edge Feature Extraction : For the edge feature, we use a vector to store the features of each edge. The edge matrix dimension is 26, which consists of system call type features,

Algorithm 5: Enriched Provenance Graph Generation

Function `GenerateEnrichedProvenanceGraph` ($\mathcal{L}$)
  Input: Log stream $\mathcal{L}$ with system/application events
  Output: Enriched heterogeneous graph $G = (V, E, A, R)$
  $V, E \leftarrow \varnothing$ `/* Initialize nodes and edges */`
  $\mathcal{M} \leftarrow$ initialize empty mappings for node deduplication;
  foreach *log entry* $l \in \mathcal{L}$ **do**
    $u \leftarrow$ `CreateTypedNode` (l.`source`, l.`type`, l.`context`)
    $v \leftarrow$ `CreateTypedNode` (l.`target`, l.`target_type`, l.`context`)
    $e \leftarrow$ `CreateTypedEdge` (u, v, l.`syscall`, l.`timestamp`, l.`args`)
    $V \leftarrow V \cup \{u, v\}, \quad E \leftarrow E \cup \{e\}$
  $G \leftarrow (V, E)$
  $G \leftarrow$ `ExtractFeatures` (G) `/* Contextual node/edge attributes */`
  $G \leftarrow$ `NormalizeGraph` (G) `/* Normalize features, add degrees */`
  return G
Function `CreateTypedNode` (*id*, *type*, *ctx*)
  if *id* $\notin \mathcal{M}$ **then**
    Create node v of given type with context;
    $\mathcal{M}$[id] $\leftarrow v$;
  return $\mathcal{M}$[id]
Function `CreateTypedEdge` (u, v, s, τ, a)
  Create typed edge $e = (u, v, s)$ with timestamp τ and attributes a;
  return e

return value, file descriptor and count values. For each edge, we store the source node ID, destination node ID and the edge features. For system call encoding, we use a one-hot encoded vector whose length is the number of system calls (d_e). We set the value to 1.0 for the corresponding system call and 0.0 for others. We have also stored the normalised timestamp value as a feature. This representation is useful to detect anomalous system call patterns, to identify the relationship between different types of system calls and to understand the system flow better.

After processing all nodes and edges, the final graph representation consists of the node feature matrix, the edge index tensor, and the edge feature matrix, which together form the input for downstream graph machine learning models. This modular and extensible design ensures that the provenance graph generator can flexibly handle heterogeneous system interactions while maintaining compatibility with GNN architectures for anomaly detection.

6.2.3 Anomaly Detection Module

At the heart of *μProvGAE* lies its anomaly detection module, which employs a specialised heterogeneous graph autoencoder (HGAE) to learn the latent representations of normal system behaviours in an unsupervised manner. This offers advantages over the supervised models used in *μProv*, particularly in environments where labeled attack data is scarce or not available.

1. Heterogeneous Graph Autoencoder (HGAE)

We have designed a Heterogeneous Graph Autoencoder (HGAE) for training the normal behaviour patterns from enriched provenance graphs in an unsupervised manner. It learns the meaningful latent representations of normal provenance graphs while preserving their heterogeneity. The 3 components of HGAE architecture are:

(i) Heterogeneous Encoder: The encoder transforms the heterogeneous enriched provenance graph G = $\langle V, E, A, R \rangle$, where V is divided into node types $\tau \in$ {process, file, socket}, into unified latent node embeddings. Each node type is passed through a combination of Graph Convolutional Network (GCN) or Graph Attention Network (GAT) layers. These type-specific layers preserve the semantic distinctions between heterogeneous node categories while learning inter-type dependencies. The layer-wise update for node type τ at layer l follows:

$$h_\tau^{(l+1)} = \sigma\left(\sum_{\tau' \in N(\tau)} \sum_{r \in R(\tau,\tau')} W_{\tau,\tau',r}^{(l)} \cdot \mathrm{AGG}\Big(\{ h_{\tau'}^{(l)}(v) \mid v \in N_r(u) \} \Big) + W_\tau^{(l)} \cdot h_\tau^{(l)} \right) \tag{6.6}$$

Where $h_\tau^{(l)}$ represents the embeddings of type τ nodes at layer l; $N(\tau)$ denotes the neighbour node types of τ; $R(\tau, \tau')$ denotes the relation (edge) types between the node type τ and τ'; $W_{\tau,\tau',r}^{(l)}$ are the learnable weight matrics for cross-type message passing; $W_\tau^{(l)}$ are self loop weights for preserving node information; AGG is an aggregate function that combines the neighborhood information of a node from different edge types and node

types; and σ is the ReLU activation function. The node features are represented as $x_\tau \in \mathbb{R}^{|V_\tau| \times d_{162}}$, where $|V_\tau|$ is the number of nodes of type τ, and d_{162} is the dimensionality of the input features for that type. Additionally, the encoder uses the sparse edge index vector $e_{\tau,\tau'} \in \mathbb{N}^{2 \times |E_{\tau,\tau'}|}$, which define the connectivity between nodes of types τ and τ' under relation r. After two stacked GCN or GAT layers per node type, the encoder outputs latent node embeddings $z_\tau \in \mathbb{R}^{|V_\tau| \times d_\tau}$, where all node types are projected into a unified τ-dimensional latent space.

(ii) Heterogeneous Decoder: The decoder takes as input the latent node embeddings $z_\tau \in \mathbb{R}^{|V_\tau| \times d_\tau}$ generated by the encoder for each node type τ. Using the embeddings, the decoder performs three key reconstruction tasks. First, it reconstructs the original node attributes by parsing z_τ through a type-specific multi-layer perception (MLP), producing reconstructed node features $\hat{f}u \in \mathbb{R}^{|V\tau| \times d_{162}}$, where d_{162} is the input feature dimension.

$$\hat{f}_u = MLP_\tau(z_u) = W_\tau^{(2)} \cdot \sigma(W_\tau^{(1)} \cdot z_u + b_\tau^{(1)}) + b_\tau^{(2)} \tag{6.7}$$

Second, to reconstruct edge attributes, the decoder concatenates the latent embeddings of node pairs $[z_u || z_v] \in \mathbb{R}^{|E_r| \times (162+162)}$ and parses them through edge-type MLP to produce $\hat{e}(u,v) \in \mathbb{R}^{|E_r| \times f_r}$, where f_r is the edge feature dimension.

$$\hat{e}_{(u,v)} = MLP_r([z_u || z_v]) = W_r^{(2)} \cdot \sigma(W_r^{(1)} \cdot [z_u || z_v] + b_r^{(1)}) + b_r^{(2)} \tag{6.8}$$

Third, to recover the graph structure, the decoder applies an inner product decoder over latent embeddings, computing the edge existence probabilities $\hat{p}(e_{u,v}^r)$ between all node pairs, resulting in an adjacency probability matrix $\hat{A}_r \in \mathbb{R}^{|V| \times |V|}$ for each edge type r.

$$\hat{p}(e_{u,v}^r) = \sigma(z_u^T \cdot W_r \cdot z_v) \tag{6.9}$$

Here, W_r is a learnable weight matrix specific to relation type r, and σ is the activation function (e.g., sigmoid) that outputs the predicted edge probability between nodes u and v. Together, these outputs enable end-to-end reconstruction of node attributes, edge attributes, and graph connectivity from the unified latent space.

(iii) Loss Function Design: We train the model using a multi-component loss function that

jointly balances the reconstruction of node features, edge features, and graph structure.

$$L_{node} = \sum_{t \in T} (1/|V_\tau|) \sum_{u \in V_\tau} ||f_u - \hat{f}_u||_2^2 \tag{6.10}$$

Where f_u and $\hat{f}_u$ are the original and reconstructed node features.

$$L_{edge} = \sum_{r \in R} (1/|E_r|) \sum_{(u,v) \in E_r} ||e_{u,v} - \hat{e}_{u,v}||_2^2 \tag{6.11}$$

Where $e_{u,v}$ and $\hat{e}_{u,v}$ are the original and reconstructed edge attributes.

$$L_{\text{struct}} = -\sum_{r \in R} \sum_{(u,v) \in V \times V} \left[y^r_{u,v} \cdot \log(\hat{p}(e^r_{u,v})) + (1 - y^r_{u,v}) \cdot \log(1 - \hat{p}(e^r_{u,v})) \right] \tag{6.12}$$

where $y^r_{u,v}$ = 1 if an edge (u, v) of type r exists, and 0 otherwise; $\hat{p}(e^r_{u,v})$ is the predicted edge probability. The final training objective combines these components:

$$\lambda_{total} = \lambda_{node} \cdot L_{node} + \lambda_{edge} \cdot L_{edge} + \lambda_{struct} \cdot L_{struct} \tag{6.13}$$

Where $\lambda_{node}, \lambda_{edge}$, and λ_{struct} are the hyperparameters to balance the different objective.

The unsupervised HGAE model in *μProvGAE* learns the intrinsic semantics and structure of benign system behaviour effectively by simultaneously reconstructing node features, edge features, and graph topology. The compact embedded representation captures non-malicious activity patterns. In the second phase, the embeddings are used by the *Online Anomaly Detection Module* to compute anomaly scores and identify anomalies in real-time. This allows for the timely detection of malicious behaviour without the need for labelled training data.

2. Online Anomaly Detection Module

The online anomaly detection module leverages the trained HGAE to identify malicious request patterns by analyzing reconstruction errors at the graph level.

(i) Graph Embedding and Reconstruction: For each incoming request to the microser-

vice application, the *μProvGAE* Graph Generator extracts its provenance graph $G = \langle V, E, A, R \rangle$, which is normalized using the same preprocessing pipeline as employed during training. This graph is then passed through the trained encoder, producing node-level embeddings z_u for each node $u \in V$. The decoder subsequently reconstructs the node features $\hat{f}u$, edge features $\hat{e}u, v$, and graph structure (edge existence probabilities) $\hat{p}(e^r_{u,v})$. The module computes the overall anomaly score by aggregating reconstruction errors across nodes, edges, and structure:

$$S(G) = w_{\text{node}} \cdot E_{\text{node}}(G) + w_{\text{edge}} \cdot E_{\text{edge}}(G) + w_{\text{struct}} \cdot E_{\text{struct}}(G) \tag{6.14}$$

Here the individual error components $E_{node}(G)$, $E_{edge}(G)$ and $E_{struct}(G)$ are the node reconstruction error, edge reconstruction error, and structure reconstruction error for the graph G.

$$E_{node}(G) = \sum_{t \in T} (1/|V_\tau|) \sum_{u \in V_\tau} \|f_u - \hat{f}_u\|_2^2 \tag{6.15}$$

$$E_{edge}(G) = \sum_{r \in R} (1/|E_r|) \sum_{(u,v) \in E_r} \|e_{u,v} - \hat{e}_{u,v}\|_2^2 \tag{6.16}$$

$$E_{\text{struct}}(G) = -\sum_{r \in R} \sum_{(u,v) \in V \times V} \left[y^r_{u,v} \cdot \log(\hat{p}(e^r_{u,v})) + (1 - y^r_{u,v}) \cdot \log(1 - \hat{p}(e^r_{u,v})) \right] \tag{6.17}$$

The raw anomaly score is then normalized using the training distribution:

$$S_{norm}(G) = (S(G) - \mu_{train})/\sigma_{train} \tag{6.18}$$

Where μ_{train} and σ_{train} are the mean and standard deviation of the anomaly scores on the training set.

(ii) Threshold Determination: To determine the detection threshold θ, we analyze the distribution of reconstruction errors on normal (benign) graphs during the training phase.

The threshold is computed as:

$$\theta = \mu_{normal} + k \cdot \sigma_{normal} \tag{6.19}$$

where k is a scaling factor selected based on the desired false positive rate.

The complete anomaly detection pipeline operates as follows: after capturing the system and application logs for a request, the system constructs the corresponding provenance graph, preprocesses and normalizes it, and encodes it using the trained HGAE model. The decoder reconstructs the graph, and the module computes the reconstruction error to derive the weighted anomaly score. This score is then normalized and compared against the threshold θ. If $S_{\text{norm}}(G) > \theta$, the request is flagged as anomalous; otherwise, it is considered normal.

3. Training

The unsupervised training leverages our proposed HGAE. The dataset for unsupervised training consists solely of provenance graphs from benign requests, which ensures the model captures generalizable patterns of normal system behaviour. For training, we split the dataset into 70% benign graphs (for training), 15% (for validation), and 15% (for testing). Importantly, no labeled attack data was used during training to validate the unsupervised anomaly detection capability of *μProvGAE*. We used the Adam optimizer with a learning rate of 0.0001 for 100 epochs, and the detection threshold was empirically determined using the reconstruction error distribution of the training set.

6.3 Proof-of-concept Implementation

Our previous research work *μProv* resolved a key challenge in attack detection research, which is that most open-source benchmarking lacks the complexity to simulate multi-stage attacks that exploit vulnerabilities commonly identified by CVES and CWES. Therefore, we develop a microservice-based application named *PicShare* (refer fig. 6.5) to emulate

Table 6.2 Injected Vulnerabilities for Attack Emulation

Label	Attack Scenario ID	Attack Type	Vulnerability Description	Affected Microservices	Expected Log Patterns
L_8	SSTI-01 CVE-2024-29686	Server-side Template Injection	Remote attacker to execute arbitrary code via a crafted payload	ProfileService	Received request for username: <%= 7 * 7 %>
L_9	PATH-TRAV-001 CVE-2019-5418	Path Traversal	Improper input sanitization allows access to files outside the intended directory and remote code execution	PhotoService	Access attempts to sensitive files, unexpected file access errors
L_{10},L_{11}	SQL-INJ-001 CVE-2014-3704	SQL Injection	SQL queries are constructed using string interpolation, allowing user input to manipulate the query structure	UserService PhotoService	SQL syntax errors, unexpected query results,
L_{12}	INS-FILE-001 CVE-2020-36112	Insecure File Upload	Improper validation of uploaded files allows remote code execution	PhotoService	Unusual file types, file handling errors

real-world vulnerabilities for distributed microservices.

6.3.1 *PicShare*: A PoC Attack Emulation Platform

The application implements an end-to-end service for uploading pictures, as well as getting recommendations and notifications on the activity of the pictures. *PicShare* is a dockerized microservice application and supports Docker Swarm for running in multiple hosts. We emulate various attacks on this application, as summarized in table 6.2.

Functionalities: In the application, users interface with a *node.js* front-end to register and log into their account. Once logged in, they can upload a photo from their user end and view a particular user's photos. The microservices are written in Python Flask, while the back-end databases consist of a relational database MySQL, persistent MongoDB instances, and a Redis DB.

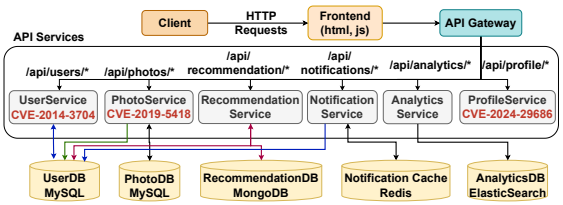


Figure 6.5 The architecture of the `PicShare` Service for uploading, sharing, and viewing photos.

It contains the following microservices: (1) a front-end server implemented using *NodeJS-EJS Templating* that serves users' requested HTML pages. It handles the application's presentation layer, facilitating dynamic content rendering. (2) *PhotoService* implemented using Python Flask to upload and view photos. This component handles the back-end processing necessary for uploading images securely and efficiently. (3) *UserService* to register and log in to the account. (4) *RecommendationService* An open-source unified analytics engine utilized for the recommendation engine of *PicShare*. (5) A *Notification-*

Service (using Flask) enables interactions related to likes, dislikes, and other user preferences. It provides a streamlined interface for communication with the recommendation engine. (7) *ProfileService* implemented using NodeJS to view the user profile. (8) *AnalyticsService* (implemented using ElasticSeach): A visualization tool integrated with ElasticDB, employed by application developers to visualize the performance metrics of the component. This application exposes 7 endpoints (*downloadphotos, getprofile, getrecommendation, loginuser, registeruser, updaterecommendation, viewphoto*) to users labeled as 'benign' $\{L_1, L_2, L_3, L_4, L_5, L_6, L_7\}$.

Attack Execution: In the context of microservices-based environments, applications are divided into smaller services, each potentially vulnerable to various types of attacks: (1) *Communication/Network Attacks*, such as Denial of Service, Man-in-the-Middle, Replay, etc., (2) *Service Level Attacks*, such as Injection (such as SQL injection and Shell Injection), Broken Access Control, Cryptographic Failures, Server-Side Request Forgery (SSRF), etc., (3) *Virtualization Attacks*, such as RCE in the host, Unauthorized Access, Container Malware, etc. We have emulated a subset of these attacks on the *PicShare* applications, as shown in table 6.2, each identified with its corresponding CVE and CWE identifiers and categorized attack types labeled as $\{L_8, L_9, L_{10}, L_{11}, L_{12}\}$. These vulnerabilities are selected based on their compatibility with the targeted versions and components of the system. In the Profile Service, we injected Malicious payloads (e.g., ⟨%= 7 * 7 %⟩) into form fields as profile name or bio, which triggered code execution through template rendering engines. The attack script issues POST requests with payloads embedded in JSON bodies or form-data fields mimicking legitimate profile updates. In PhotpService, we used file paths like `../../etc/passwd` in image download or view endpoints to access files outside the designated photo directory. The attack scripts use GET requests to */api/photos/view?file=...* with path traversal patterns and log resulting errors or leaked content. In User and Photo Service, we injected strings like "*' OR 1=1* " and "*UNION SELECT*" queries via user login or search endpoints to manipulate backend SQL queries. In Photo Service, we uploaded files with disallowed extensions (e.g., .php, .exe) or obfuscated MIME types to bypass validation. Each attack scenario was executed in a controlled environment, and the logs generated were used to construct enriched provenance graphs. These attacks expose observable patterns such as malformed file access, SQL errors, or injected payloads, which our system uses for anomaly detection. The complete set of attack scripts

and execution configurations is available in our GitHub repository[2].

6.3.2 Supervised Attack Detection Techniques

While our preliminary contribution focuses on unsupervised anomaly detection using the proposed *μProvGAE* framework, we also incorporate supervised learning models in our evaluation for two important purposes : (1) baseline comparison to assess the effectiveness of enhanced logging and provenance enrichment (as shown in section 6.4.3), and (2) fine-grained behavior modeling to understand the relationship between different attack classes and benign activities. To support these objectives, we leverage the **PicShare** application as the common data generation platform for both supervised and unsupervised experiments. This enables a controlled, uniform environment for evaluating logging quality, provenance graph expressiveness, and anomaly detection accuracy across models and techniques.

Multiclass Framing for Supervised Detection: User requests exhibit diverse behaviours that trigger distinct system call sequence signatures during execution. Our objective is to analyse these interaction patterns and signatures to differentiate between benign and malicious requests, as well as between different types of malicious requests. Given that user-facing applications often include multiple operations (or endpoints), we treat each request type as a separate class, resulting in 12 different types. This allows us to approach attack detection as a multiclass classification problem with 7 benign and 5 malicious request classes.

Feature Extraction: We extract relevant features from both log data $\mathcal{L}$ and the provenance graphs $\mathcal{G}$. Initially, we construct a provenance graph from the global log file for each request, resulting in a set of graphs G_i, where each graph is labeled according to its respective class as either benign or attack, with its specific type $L_i \in \{L_1, L_2, \ldots, L_{12}\}$ (refer to section 5.3). From each graph G_i, we derive basic graph-related features, such as the number of nodes, edges, average degree, density, degree centrality, number of connected components, node connectivity, edge connectivity, in-degree and out-degree centrality, and degree associativity. Additionally, we extract specialized features from the provenance graphs, in-

[2] https://github.com/usatpath01/MuProv (Accessed: May 5, 2026)

cluding the number of system calls, system call frequencies, system call counts, and their return values. While graph-related features help us efficiently detect malicious requests, provenance-specific features allow us to identify subtle changes in system call sequences that could indicate small-scale attacks, which might otherwise resemble benign requests.

Supervised Model Selection and Training: We explore standard supervised multiclass classification techniques commonly employed in traditional frameworks for detecting various benign and attack scenarios [47, 50]. Specifically, we investigate classification models such as K-Nearest Neighbors (KNN) [141], Support Vector Machines (SVM) [88, 141], Random Forest [12], and Artificial Neural Networks (ANN) [142], as these models have been shown to perform well with high-dimensional numerical features. We implement versions of these ML models for the multiclass classification task, training each model with labelled features extracted from the graphs. All models were trained on the same set of features and their vectorized representations. Specifically, for each graph G_i along with the corresponding label L_i, we extract all relevant features and represent them as a one-dimensional vector, where the length of the vector corresponds to the total number of features extracted. These feature vectors are then used as input for training and testing the attack detection models. For KNN, the number of neighbours, k = 5, the distance metric is Euclidean. For SVM, we used the kernel as Radial Basis Function (RBF), C = 1.0. The Random Forest model is configured with a number of trees of 100 and a minimum sample split of 2. In ANN, we have configured 2 hidden layers (128 and 64 neurons) with activation function as ReLU and Adam Optimizer. The learning rate is 0.0001, the epoch size is 100, and the batch size is 32. The models are trained using a 70-15-15% train-validation-test split to assess the capabilities. The purpose of this supervised setup is not to claim novelty in classification algorithms, but to demonstrate the effect of μ*Prov*'s logging quality and enriched provenance representation on traditional detection pipelines. Hence, this section serves two purposes: as a benchmark layer for evaluating the impact of our logging and graph modelling and secondly, by enabling direct comparisons with μ*ProvGAE* to highlight the performance gains enabled by unsupervised anomaly detection on semantically enriched provenance data.

6.4 Evaluation

In this section, we present a comprehensive evaluation of *μProvGAE*, addressing two distinct comparison frameworks: (1) logging framework effectiveness comparing Tracee vs *μProv* [143] (our prior work) using supervised machine learning models, and (2) anomaly detection framework performance comparing *μProvGAE* against state-of-the-art-graph-based detection methods. Our evaluation demonstrates the effectiveness of both enhanced logging and unsupervised graph encoders architecture for attack detection in distributed microservice environments.

Our evaluation aims to answer the following question:

- **RQ1:** How do the causally ordered logs generated by *μProv* and *μProvGAE* impact the construction of provenance and enriched provenance graphs compared to `Tracee`[3], a widely used eBPF-based system auditing framework? Additionally, how does the graph enrichment influence detection performance? (Refer table 6.3)
- **RQ2:** How accurately can *μProvGAE* detect anomalies, and how do its detection performance metrics (precision, recall, accuracy) compare against state-of-the-art baselines? (Refer table 6.4)
- **RQ3:** What is the contribution of each component in the Heterogeneous Graph Autoencoder (HGAE) to overall detection performance, and how does each affect the system's ability to model complex graph patterns? (Refer table 6.5)

6.4.1 Baselines

Our evaluation strategy involves two complementary comparison frameworks, one focused on log collection quality and the other on graph-based anomaly detection. Each requires carefully chosen baselines aligned with its specific objectives to ensure fair and meaningful evaluation.

[3] https://github.com/aquasecurity/tracee (Accessed: May 5, 2026)

Logging Framework Comparison: *μProv* vs. Tracee To assess the impact of our enhanced logging architecture, we compare *μProv* with Tracee, a widely adopted, open-source eBPF-based system auditing framework. Tracee serves as a strong representative of current best practices in Linux container security, offering extensive syscall monitoring capabilities without application-level context integration. In this setting, our goal is to evaluate the effect of provenance graph quality and semantic richness on detection performance. Accordingly, we apply the same suite of classical supervised learning models: K-Nearest Neighbours (KNN), Support Vector Machines (SVM), Random Forest, and Artificial Neural Networks (ANN), to both *μProv* and Tracee datasets. This ensures that any performance differences are attributable solely to the logging framework and the resulting graph representations, not to differences in classifier capabilities.

Anomaly Detection Framework Comparison: *μProvGAE* vs. State-of-the-Art Baselines For evaluating graph-based anomaly detection, we compare *μProvGAE* against two state-of-the-art provenance-based methods: UNICORN [12] and ProvDetector [107]. These baselines represent both supervised and unsupervised paradigms. UNICORN is a supervised runtime provenance analysis system designed for Advanced Persistent Threat (APT) detection.It utilises runtime provenance-based attack detection for APTs using a Random Forest Machine Learning Model on graph features. ProvDetector [107] represents an unsupervised approach using infrequent path detection and Local Outlier Factor (LOF) scoring. It is explicitly designed to detect stealthy or novel attacks without relying on labeled training data, making it a strong methodological counterpart to *μProvGAE* in scenarios where labeled attack instances are unavailable.

By maintaining this separation, i.e., comparing *μProv* vs. Tracee under the same classifiers, and *μProvGAE* vs. UNICORN and ProvDetector within a graph-based detection context, we ensure that enhancements in logging fidelity and anomaly detection methodology are independently evaluated. This dual-framework approach allows us to demonstrate both the upstream benefits of semantic log enrichment and the downstream effectiveness of unsupervised graph-based learning for novel attack detection.

Table 6.3 Performance Comparison for our Framework

Model Name	**Tracee**				**μProv**			
	Accuracy	Precision	Recall	Micro-F1 Score	Accuracy	Precision	Recall	Micro-F1 Score
KNN	48.27	47.76	46.66	46.18	70.36	70.39	71.21	70.61
SVM	38.69	42.23	40.18	39.36	83.14	81.14	80.25	80.12
Random Forest	52.80	54.93	52.80	53.10	87.78	87.97	87.78	87.64
ANN	46.32	56.54	48.81	48.72	83.05	89.90	90.18	88.21

6.4.2 Implementation Details

The source code, documentation, and experimental configuration scripts for our implementation have been open-sourced[4]. For *μProvGAE*'s Logging Framework, we used C with the *libbpf* library to create eBPF probe programs, identifying 21 configurable syscalls for which we developed separate eBPF probes. For the kernel-space component, we set the eBPF ring buffer size to 1MB with a 50ms polling interval, forwarding logs to the Logging collector. Additionally, we reserved 2MB for the $exec\_map$ and 64KB for the $args\_map$. Provenance Graph Generator is a Python program with approximately 880 lines of code. We have implemented *μProvGAE*'s heterogeneous enriched graph autoencoder (HGAE) using PyTorch and PyTorch Geometric, with 2 graph convolutional layers (hidden dimension 128, latent dimension 64), ReLU activations and dropout regularization (p = 0.3).

The system was evaluated on the **PicShare** microservice application, a containerized photo-sharing platform with 13 distinct microservices designed to simulate real-world benign and attack scenarios. The main user interaction endpoints include the *UserService (US)*, *PhotoService (PS)*, *RecommendationService (RS)*, and *ProfileService (PS)*. We developed an automation script that generates 1000 requests, spaced 5 seconds apart, targeting both benign and attack scenarios across services. The log dataset includes 12 classes: 7 benign and 5 attack scenarios as labelled as L_1 to L_{12} generated by injecting known vulnerabilities (with corresponding CVE and CWE identifiers) into specific services (refer table 5.1). To evaluate the framework's ability to collect logs across multiple hosts, we deployed the microservices on 10 virtual machines (VMs) configured as edge computing hosts. Each VM hosted several containerized microservices, managed by Docker Swarm, and was equipped with Ubuntu 22.04 SMP, the Linux 6.5.0-41-generic kernel, 16 vCPU cores, and 64GB of RAM. One VM serve as the Docker Swarm manager, while the re-

[4] https://github.com/usatpath01/MuProvGAE (Accessed: May 5, 2026)

maining VMs operated as worker nodes.

We evaluated system performance using standard metrics: accuracy, precision, recall, and Micro-F1 score. To assess practical effectiveness, we also examined the false positive rate (FPR) and false negative rate (FNR), as well as resource usage (CPU and memory overhead) compared to the baselines. FP is when a benign event is misclassified as an attack, or one type of attack is misclassified as another. FN is when an attack is classified as benign activity or a different attack.

6.4.3 Results

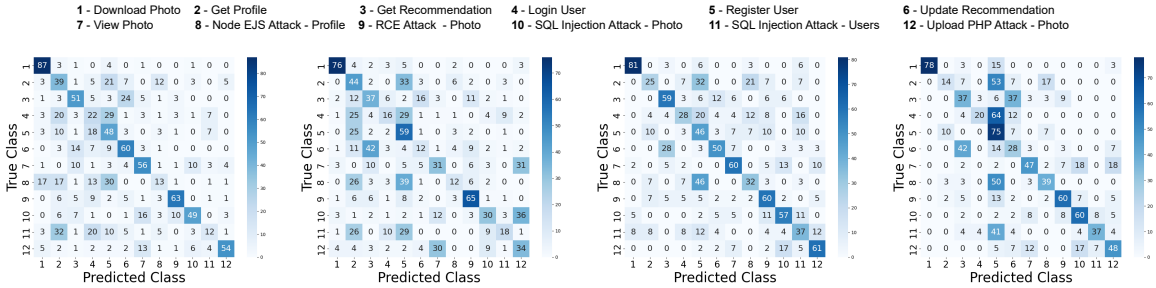
This section presents a comprehensive evaluation of our system through two key experiments: (i) supervised attack detection comparing Tracee and *μProvGAE* using traditional machine learning models, and (2) unsupervised anomaly detection with *μProvGAE*, evaluated against state-of-the-art baselines. We report quantitative results, technical insights, and key findings from both evaluations.

1. Logging Framework Effectiveness

This section evaluates the quality and utility of system logs for fine-grained attack detection by comparing the performance of two logging frameworks: Tracee and *μProv*, under a controlled experimental setup. To ensure a fair comparison, we hold the classification algorithms constant and vary only the logging source, thereby isolating the effect of log fidelity and semantic richness on detection performance.

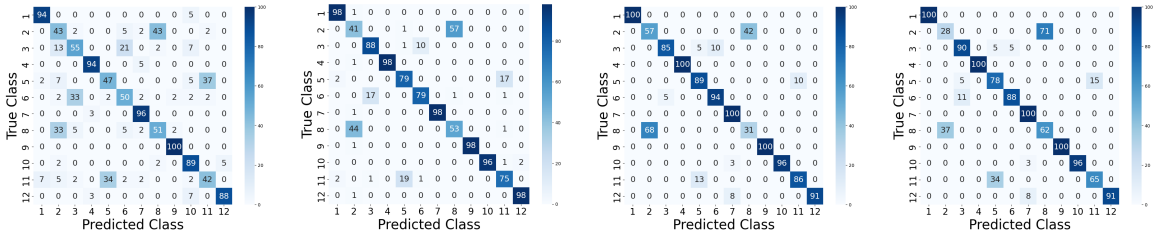
(i) Supervised Classification Performance

(a) Quantitative Comparison of Logging Quality: Table 6.3 presents the supervised classification results using four standard machine learning models: KNN, SVM, Random Forest, and Artificial Neural Network (ANN). Each model is trained and evaluated independently on datasets generated from Tracee and *μProv*, ensuring that performance differences are attributable solely to the quality of the provenance graphs derived from the



(a) Tracee - KNN **(b)** Tracee - SVM **(c)** Tracee - Random Forest **(d)** Tracee - ANN

Figure 6.6 Confusion Matrix for Tracee (%)



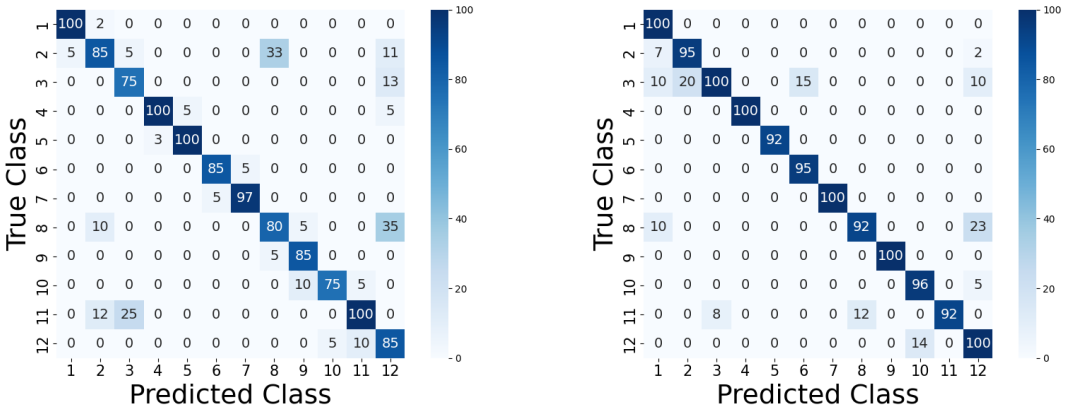
(a) $\mu Prov$- KNN **(b)** $\mu Prov$- SVM **(c)** $\mu Prov$- Random Forest **(d)** $\mu Prov$- ANN

Figure 6.7 Confusion Matrix for $\mu Prov$ (%)

respective logging frameworks. Results on Tracee data reveal significant limitations in capturing discriminative features necessary for attack detection. Random Forest achieves the best performance among the four models, with 52.80% accuracy and 53.10% Micro-F1 score. However, both KNN and SVM struggle significantly, producing Micro-F1 scores of 46.18% and 39.36%, respectively. These outcomes suggest that raw system call sequences, as captured by Tracee, lack the semantic granularity required to differentiate between closely related benign and malicious activities. In contrast, the use of $\mu Prov$, which generates causally ordered and semantically enriched logs, results in substantial improvements across all classifiers. Random Forest performance improves dramatically, reaching 87.78% accuracy and 87.64% Micro-F1 score, reflecting a 34.98 percentage point increase in accuracy over Tracee. The ANN model shows the most pronounced gain, improving from 48.72% to 88.21% Micro-F1, with a recall of 90.18%. Even traditionally weaker classifiers exhibit substantial boosts: SVM rises from 39.36% to 80.12% Micro-F1, and KNN increases from 46.18% to 70.61%, demonstrating the value of enriched logging even for less complex models.

(ii) Classification Quality Analysis

(a) Qualitative Assessment via Confusion Matrices: Figure 6.6 and 6.7 visualize the confusion matrices for classifiers trained on Tracee and *µProv* logs, respectively. Tracee-based models (fig. 6.6) exhibit extensive off-diagonal confusion, particularly between functionally similar classes. For instance, there is notable misclassification between Class 2 ("Get Profile") and Class 3 ("Get Recommendations"), as well as between Class 8 ("Node.js EJS attack") and Class 9 ("RCE attack on photo"). These errors indicate that Tracee's syscall-based logs are insufficient to differentiate nuanced behavioral patterns across microservice interactions. In contrast, *µProv* based models (fig. 6.7) exhibit clear diagonal dominance, indicating precise class boundaries and robust pattern recognition. Both Random Forest and ANN achieve near-perfect classification for most classes, confirming that *µProv*'s enriched logs preserve behavioral semantics and causal dependencies essential for distinguishing benign and malicious request flows, even across structurally similar activities.

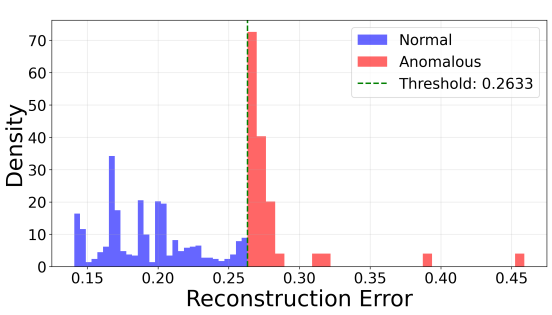


Old Provenance Graph + GNN + Supervised New Provenance Graph + GNN + Supervised

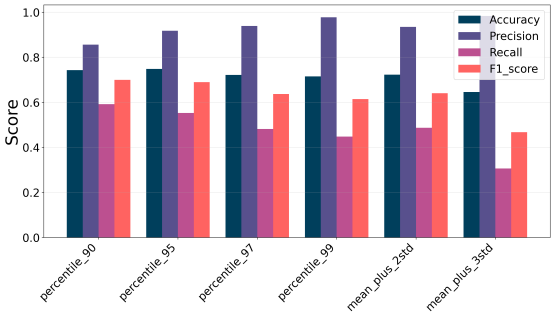
Figure 6.8 Confusion Matrix of Attack Detection using Old vs Enriched Provenance Graph

(b) Impact of Semantic Graph Enrichment: To further evaluate the effect of provenance graph enrichment, we compare supervised GNN models trained on traditional and enriched graphs using identical network architectures. As shown in fig. 6.8, the enriched provenance graph derived from *µProv* logs enables significantly improved class separability—particularly for attack scenarios that exhibit overlapping structural patterns in traditional graphs. These findings validate our semantic feature integration strategy, which augments graph representations with rich node attributes, edge semantics, and execution

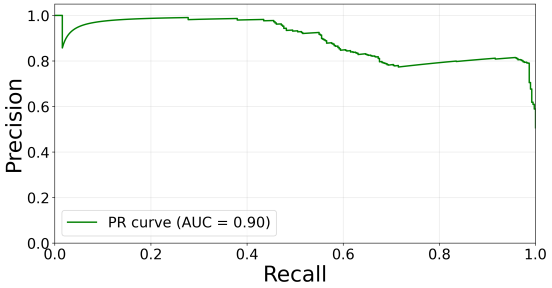
context, thereby enhancing model interpretability and classification effectiveness.



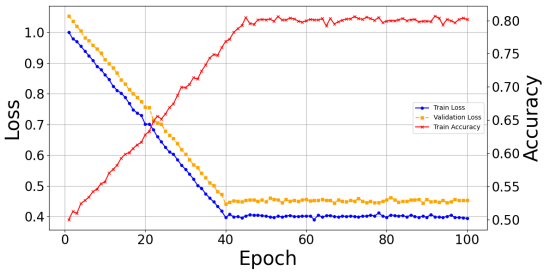
(a) Anomaly Score Distribution



(b) Performance Comparison by Threshold Method



(c) Precision - Recall Curve



(d) Loss and Accuracy

Figure 6.9 Anomaly Detection Performance - Training Progress Over 100 Epochs

2. Anomaly Detection Framework Performance

This section evaluates the unsupervised anomaly detection capabilities of *μProvGAE* in comparison to state-of-the-art graph-based detection techniques.

(i) Unsupervised Detection Performance:

(a) Quantitative Comparison Across Frameworks: Table 6.4 provides a detailed comparison of unsupervised anomaly detection performance across three methods—*μProvGAE*, UNICORN, and ProvDetector—evaluated on both traditional and semantically enriched provenance graphs. This analysis isolates the effectiveness of anomaly detection algorithms independent of logging or labeling assumptions.

On traditional provenance graphs, UNICORN achieves the highest performance, with a Micro-F1 score of 81.70%, precision of 79.18%, and recall of 78.27%. These results reflect

Table 6.4 Overall Detection Performance

Method	Old Provenance Graph			Enriched Provenance Graph		
	Micro F1-Score	Precision	Recall	Micro F1-Score	Precision	Recall
UNICORN [12]	81.70	79.18	78.27	83.70	82.75	90.12
ProvDetector [107]	43.57	61.51	36.25	47.57	66.05	37.17
μProvGAE	74.31	83.73	58.14	85.27	95.61	69.16

UNICORN's advantage as a supervised learning framework, utilizing graph sketch representations optimized for structural features. In comparison, *μProvGAE* attains a Micro-F1 of 74.31%, highlighting the inherent challenge of purely unsupervised detection on limited structural data. However, when evaluated on enriched provenance graphs, the performance dynamics shift significantly. *μProvGAE* outperforms UNICORN, achieving a Micro-F1 score of 85.27%, with precision of 95.61% and recall of 69.16%. This reversal underscores *μProvGAE*'s ability to exploit the richer semantic context embedded in the enriched graphs, effectively capturing nuanced behavioral anomalies that escape purely structural models. UNICORN, while still competitive, reaches a slightly lower F1 score of 83.70% and precision of 82.75%, demonstrating that its performance gains plateau without access to semantic augmentation. In contrast, ProvDetector performs poorly across both graph types, achieving only 43.57% Micro-F1 on traditional graphs and a marginally improved 47.57% on enriched graphs. The substantial 37.7 percentage point gap between *μProvGAE* and ProvDetector on enriched graphs highlights the inadequacy of traditional outlier-based approaches—such as those using Local Outlier Factor (LOF)—in capturing high-dimensional, semantically complex patterns inherent in modern microservice-based security datasets.

Table 6.5 Contribution of different components

Variant	Description	Accuracy	Micro F1-Score
μProvGAE	Full Model	85.78	85.27
- Node Features	Without node feature reconstruction	79.50	78.98
- Edge Features	Without edge feature reconstruction	77.61	77.32
- Structure Loss	Without structural reconstruction loss	72.65	71.54

(ii) Detection Quality and Threshold Analysis:

(a) Anomaly Score Distribution and Threshold Selection: Figure 6.9a illustrates the distribution of anomaly scores produced by *μProvGAE* across benign and attack graphs. Benign graphs exhibit tightly clustered scores in the range of 0.11 to 0.25, while attack

graphs span a broader and higher range of 0.26 to 0.45, indicating a clear separation between normal and anomalous behaviors. This separation enables the application of an effective thresholding strategy for binary classification. Through empirical evaluation, we determined the optimal decision threshold as: $\theta = \mu_{\text{train}} + 2.5 \times \sigma_{\text{train}} = 26.33$, which yields a low false positive rate of approximately 2.9%, making it well-suited for deployment in high-assurance security environments.

(b) Precision-Recall Performance and Threshold Robustness: As shown in fig. 6.9c, *μProvGAE* achieves a Precision-Recall AUC of 0.90, demonstrating strong discriminatory capability even under class imbalance, which is an expected characteristic in security-sensitive domains. Figure 6.9b further confirms the effectiveness of our thresholding methodology. The statistical rule of setting the threshold at $(mean \pm 2.5)$ standard deviations consistently delivers optimal trade-offs between precision and recall across multiple validation runs and attack variants, validating its robustness for real-world applications. While high recall is desirable to minimize missed attacks, high precision reduces alert fatigue and analyst workload. Hence, *μProvGAE* demonstrates higher precision, making it suitable for environments where reducing false positives is critical.

(c) Training Convergence Behavior: The training dynamics of *μProvGAE* are depicted in fig. 6.9d, which shows the progression of reconstruction loss for node features, edge features, and structural consistency over 100 training epochs. All components exhibit smooth convergence, with loss values stabilizing after approximately 60 epochs, indicating effective learning and reliable generalization. Notably, there is no evidence of overfitting, as the validation loss mirrors the training loss throughout the process, confirming the suitability of our architectural and regularization choices for unsupervised anomaly detection in complex graph-structured data.

3. Component Effectiveness Analysis

(i) HGAE Architecture Ablation Study:

(a) Individual Component Contributions: To evaluate the relative importance of each reconstruction objective in the *μProvGAE* architecture, we conduct a detailed ablation study as summarized in table 6.5. The full model, incorporating node feature, edge feature,

and structural topology reconstruction, achieves 85.78% accuracy and a Micro-F1 score of 85.27%. Ablation of the node feature reconstruction component results in a performance drop to 79.50% accuracy and 78.98% Micro-F1, reflecting a 6.28% decrease in accuracy and 6.29% in F1. Removing edge feature reconstruction causes a more significant decline to 77.61% accuracy and 77.32% Micro-F1, corresponding to 8.17% and 7.95% reductions, respectively. The most substantial degradation is observed when the structural reconstruction loss is omitted, reducing the model's performance to 72.65% accuracy and 71.54% Micro-F1, a drop of 13.13% in accuracy and 13.73% in F1 score.

These results clearly indicate that all three reconstruction components are essential for robust anomaly detection. Among them, structural reconstruction plays the most critical role, followed by edge and node features. Importantly, the cumulative impact of all components exceeds the linear sum of their individual contributions, suggesting synergistic interactions among heterogeneous reconstruction objectives that enable the model to capture multi-faceted graph anomalies more effectively.

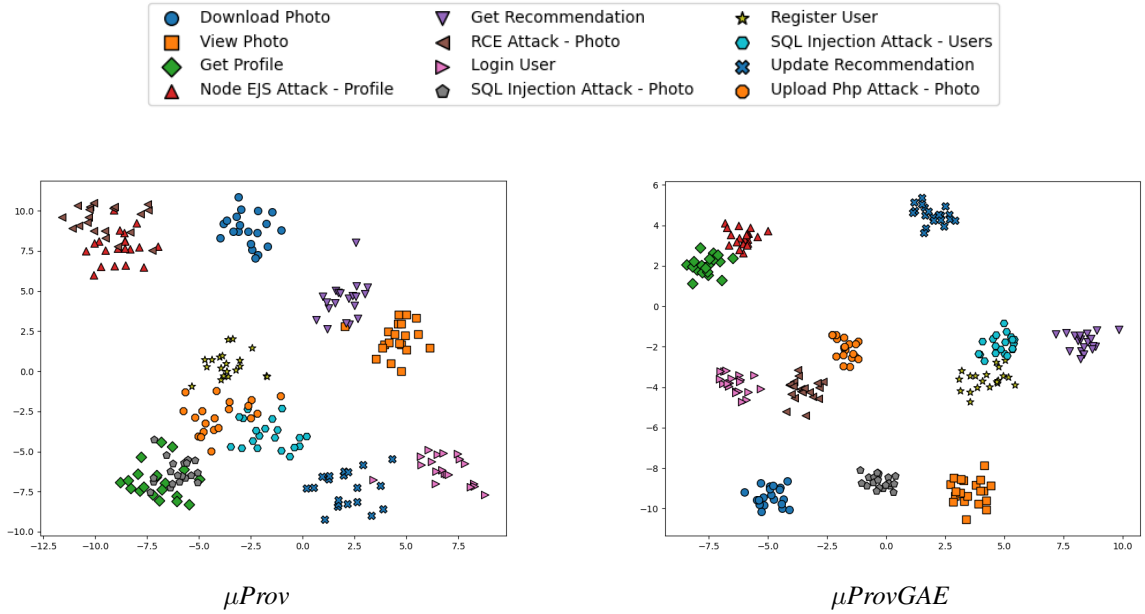


Figure 6.10 Embedding visualization of PicShare dataset. Different colors indicate different node category labels. PCA → KMeans Clustering: 12 Classes in Different Clusters (20 Samples Each)

(b) Embedding Quality Analysis: To assess the quality of the latent representations

learned by *μProvGAE*, we perform a Principal Component Analysis (PCA) projection of the graph embeddings across all 12 PicShare service classes. As visualized in section 6.4.3, embeddings derived from the *μProv* (non-semantic) framework exhibit significant cluster overlap and poor separability. In contrast, section 6.4.3 demonstrates that *μProvGAE* embeddings form well-separated, compact clusters corresponding to individual classes, with minimal inter-class confusion. This distinct cluster formation validates the model's capacity to learn semantically meaningful and discriminative embeddings. The results reinforce the architectural effectiveness of HGAE in preserving functional semantics and behavioral signatures necessary for accurate anomaly detection in complex, heterogeneous graph data.

6.4.4 Research Question Analysis

Our experimental evaluation directly addresses the three central research questions outlined in this study:

RQ1: What is the impact of causally ordered logs and semantic graph enrichment on detection performance? As shown in table 6.3, *μProv*'s causally ordered and semantically enriched logging framework significantly outperforms Tracee across all supervised machine learning models. The average accuracy improvement ranges from 30 to 35 percentage points, highlighting the value of preserving causality and application-level context in log data. Furthermore, fig. 6.8 demonstrates that provenance graph enrichment via semantic augmentation of node and edge features yields additional performance gains over traditional structural graph representations. These results validate both components of our logging framework: the causal ordering of system and application events, and the integration of semantic information into the graph structure.

RQ2: How accurate is *μProvGAE* in detecting anomalies compared to state-of-the-art methods? As reported in table 6.4, *μProvGAE* achieves a Micro-F1 score of 85.27% on enriched provenance graphs, outperforming UNICORN (83.70%) and substantially surpassing ProvDetector (47.57%). In addition, *μProvGAE* attains a precision of 95.61%, indicating strong reliability and low false positive rates, an essential requirement for production-grade security systems. These results establish *μProvGAE*'s superiority in detecting complex and previously unseen attack patterns under unsupervised settings.

RQ3: What is the relative contribution of each component in the *μProvGAE* architecture? The ablation study results in table 6.5 demonstrate the individual and combined effects of the model's reconstruction components. The structural reconstruction loss has the highest impact, contributing 13.13 percentage points to accuracy, followed by edge feature reconstruction (8.17%) and node feature reconstruction (6.28%). Notably, the cumulative performance exceeds the sum of individual contributions, indicating synergistic interactions among architectural components. These findings validate the design of our heterogeneous graph autoencoder as an effective means to capture diverse and interrelated patterns in multi-dimensional graph-structured security data.

6.5 Summary

In this chapter, we developed an advanced anomaly detection system known as *μProvGAE*, designed to detect Advanced Persistent Threats (APTs) in distributed microservice environments. *μProvGAE* extends our prior work, μProv, by introducing enriched heterogeneous provenance graphs and a specialized heterogeneous graph autoencoder (HGAE) for unsupervised anomaly detection at runtime. Our system captures causally ordered, fine-grained logs across microservices using an eBPF-based logging framework. These logs are processed into semantically rich provenance graphs, representing diverse system entities and their interactions. *μProvGAE* learns normal behavioral patterns using unsupervised graph learning, enabling it to flag anomalies without relying on labeled attack data. We implement a multi-stage architecture consisting of data collection, graph generation, and anomaly detection. The anomaly score is computed using reconstruction errors across nodes, edges, and graph structure. To demonstrate *μProvGAE*'s practical relevance, we evaluate it using a custom photo-sharing microservice application (PicShare) deployed across 10 hosts, emulating real-world vulnerabilities such as SQL injection, insecure file upload, and server-side template injection. *μProvGAE* achieves 85.27% F1-score with a 95.61% precision rate, outperforming both classical supervised baselines and state-of-the-art graph-based models such as UNICORN and ProvDetector. Notably, *μProvGAE* operates in real-time and adapts to evolving attack patterns. This work demonstrates the feasibility and scalability of semantically enriched, graph-based anomaly detection in modern microservice architec-

tures. While *μProvGAE* is achieving strong performance at the evaluated scale, extending the system to significantly larger deployments introduces scalability challenges primarily related to provenance data volume and graph construction overhead, rather than limitations of the graph autoencoder itself. As the number of services, hosts, and concurrent requests increases, the generation, storage, and processing of fine-grained provenance logs grow rapidly, making provenance management the primary bottleneck. Addressing this bottleneck provides a promising area for future research. The current design partially achieves scalability by segmenting the provenance graphs at the request level and utilizing causality-aware filtering. This guarantees that only relevant graphs are carefully scrutinized and that the effective graph size remains manageable. To improve scalability for large-scale scenarios, various system-level enhancements are planned for future versions. These features involve reducing the volume of provenance data, filtering out security-related system calls, updating graphs when changes occur, and using multi-tier processing pipelines for initial processing at the source and analysis at the center. These strategies promise a roadmap for scaling the framework to large-scale enterprise scenarios without compromising detection effectiveness.

Chapter 7

Conclusion and Future Work

Modern cloud-native systems built on containerized microservice architectures introduce significant challenges in terms of security observability, attack provenance tracking, and real-time anomaly detection. With the growing scale and complexity of such applications, especially in edge and distributed environments, traditional logging and monitoring tools fall short in preserving causality, ensuring contextual correlation across distributed logs, and enabling scalable forensic analysis. This thesis presents a comprehensive pipeline for secure, scalable, and real-time attack investigation and anomaly detection by leveraging causally ordered logs and provenance graphs. Our contributions span from lightweight instrumentation-free logging systems to graph-based learning models for accurate anomaly detection.

In this thesis, we develop a series of systems: *DisProTrack*, *XPLOG*, μ*Prov*, and μ*ProvGAE*—that progressively build a causality-preserving, provenance-aware, and graph-driven foundation for distributed attack investigation. These systems work synergistically to address the limitations of prior works, such as lack of distributed log causality, coarse-grained observability, and reliance on labeled data for attack detection. We also introduce a novel benchmark dataset through *PicShare* to support reproducible experimentation in this domain. Together, these efforts form a robust and scalable framework for security analytics in distributed microservices.

7.1 Summary of Contributions

This thesis targets key stakeholders involved in securing and managing distributed microservices namely developers, system administrators, cloud operators, and security analysts. Below, we summarize how each of the contributions addresses core challenges in provenance tracking, forensic analysis, and anomaly detection.

7.1.1 *DisProTrack*: Distributed Provenance Tracking over Serverless Applications

DisProTrack is a lightweight, application-agnostic system designed to reconstruct execution traces and identify the root cause of anomalies in serverless and distributed environments. It tackles the inherent challenge of correlating system-level and application-level logs without source-code instrumentation. By combining static analysis (LMS-CFG) and a runtime Loadable Kernel Module (LKM) that intercepts write syscalls, *DisProTrack* constructs a Universal Provenance Graph (UPG) with high fidelity. Real-world evaluations on benchmark serverless apps demonstrate its effectiveness in detecting complex attacks like data exfiltration with minimal performance overhead. The system completes provenance graph reconstruction within 20–30 seconds, enabling near real-time analysis.

DisProTrack demonstrated that just using logs collected separately to piece together the history of actions later on is not enough to accurately understand how things work from start to finish in distributed systems. The failure to maintain causality across hosts led to disjointed execution paths and limited security visibility. This lesson directly inspired *XPLOG*, which enforces causal ordering when logs are created. This ensures that provenance data is consistent, complete, and ready for analysis later.

7.1.2 *XPLOG*: A Dynamic Observability Framework for Distributed Sandboxed Microservices

To address the limitations of trace disorder and disjoint logs in distributed microservices, *XPLOG* introduces a causality-aware observability framework using eBPF. *XPLOG* intercepts system and application-level events, enriching them with contextual metadata and maintaining execution ordering through vector clocks. It provides a unified and expressive log format that significantly improves traceability and enables efficient query filtering. Evaluated with the DeathStarBench benchmark, *XPLOG* demonstrates a marked improvement in causal consistency and query efficiency over existing tools such as Tracee, while maintaining low CPU, memory, and network usage making it suitable for edge deployments.

XPLOG enabled collecting logs from many hosts in a scalable, causally ordered way, but it became clear that causal logs alone are not sufficient to detect attacks. The volume and complexity of analyzing distributed logs by hand or with rules were still too high. This idea led to μ*Prov* turning causally ordered logs into structured provenance graphs. This lets us reason about distributed executions more comprehensively using graphs.

7.1.3 μ*Prov*: Provenance Graph Generation for Attack Detection

Building upon *XPLOG*'s enriched logs, μ*Prov* constructs fine-grained, dynamic provenance graphs to support real-time attack detection in microservice environments. By capturing file access, process interaction, and network flows, μ*Prov* creates a comprehensive system interaction graph. A benchmark dataset (PicShare) was introduced to evaluate the approach with both benign and malicious traces. Traditional ML classifiers trained on provenance features achieve high accuracy, with Random Forest models reaching 87.78% detection accuracy while halving detection latency. μ*Prov* also reduces false positives compared to baseline logging methods, demonstrating the efficacy of graph-based context modeling for attack investigation.

The μ*Prov* study showed that provenance graphs make it much easier to detect attacks,

but it also demonstrated that manual feature engineering and rule-based detection don't work well for complex or evolving attacks. This lesson inspired the creation of *μProvGAE*, a graph autoencoder that learns normal behavior directly from provenance structures. This means it doesn't have to rely on hand-crafted features as much and can instead leverage the causally consistent graphs that it *μProv* generates.

7.1.4 *μProvGAE*: Graph Autoencoder for Unsupervised Anomaly Detection

To address the lack of labeled data and the evolving nature of attacks, *μProvGAE* extends *μProv* by leveraging heterogeneous graph autoencoders for unsupervised anomaly detection. It models diverse system entities and interactions as a semantically enriched heterogeneous provenance graph. Using reconstruction errors across nodes and edges, *μProvGAE* identifies anomalous behavior without labeled data. Evaluations using real-world multi-stage attack scenarios on PicShare yield an F1-score of 85.27% and precision of 95.61%, outperforming state-of-the-art baselines such as UNICORN and ProvDetector. This system demonstrates the feasibility of scalable and adaptive graph-based security analytics.

The work as a whole shows that each layer: causal logging, provenance construction, and learning-based detection, needs the previous layer to be right and complete. Finding better ways to gather causal data directly improved provenance quality, which in turn enabled more precise anomaly detection and practical use at the operations level. This layered progression shows that strong security analytics in distributed microservices need to be built up over time, with strong guarantees at each step, instead of relying on separate detection methods

7.2 Directions of Future Work

In this section, we consider some of the possible future directions that have opened up with the insights from this thesis.

7.2.1 Extending to Heterogeneous Infrastructure and DevSecOps Pipelines

While the proposed logging and provenance systems have been demonstrated in containerized microservice environments, production deployments often span heterogeneous infrastructures, including bare-metal servers, VMs, containers, and serverless functions across edge and multi-cloud platforms. A key direction is to ensure that provenance logging, anomaly detection, and policy enforcement operate consistently across these diverse execution substrates. Adapting the eBPF-based logging used in *XPLOG* and *μProv* to such platforms requires addressing syscall interface differences (e.g., runc, gVisor, Firecracker), tracing limitations in serverless environments (e.g., AWS Lambda), and kernel-level constraints in unikernels. Recent systems like Sysfilter [144] and Timeloops [145] show promise for syscall policy generation, but dynamic, causality-preserving integration remains an open challenge. Incorporating provenance analysis into DevSecOps workflows can further shift security left in the development lifecycle. This includes: (i) embedding provenance checks and syscall profiling into CI/CD pipelines; (ii) auto-generating least-privilege syscall policies (e.g., Seccomp/AppArmor) during container builds [146]; (iii) leveraging anomaly detection over historical traces before deployment; and (iv) enforcing runtime policies via OPA or Kyverno.

Furthermore, Infrastructure-as-Code tools (e.g., Terraform, Pulumi) can use provenance insights to configure network rules, sandboxing, and resource access. Such cross-layer integration will enable provenance systems to evolve from reactive forensic tools into proactive components within secure, automated software delivery pipelines—critical for ensuring compliance, auditability, and resilience in complex deployments.

7.2.2 Compression Schemes for Provenance Logs in High-Throughput Environments

As microservice-based systems scale across distributed infrastructures, the volume of fine-grained, causally ordered provenance logs (e.g., from *XPLOG* or *μProv*) can grow rapidly, posing significant challenges for storage, transmission, and synchronization—especially in edge environments. To address this, future work should explore provenance-aware com-

pression techniques that go beyond generic methods like Gzip or LZ4 and leverage domain-specific structure and semantics. Promising directions include: (1) Structure-Aware Compression: Use dictionary encoding, delta encoding, and grammar-based compression (e.g., Sequitur) to exploit repeated patterns in syscall names, timestamps, and container IDs; (2) Causality-Preserving Compression [147]: Ensure that compression maintains vector clock order to preserve causal relationships across distributed logs; (3) Graph-Based Delta Encoding [148]: Compress only the incremental changes (e.g., added nodes or edges) in provenance graphs to reduce redundancy; (4) Query-Aware Summarization: Tailor compression based on query patterns, using data structures like Bloom filters or compressed indexes to retain relevant events; (5) Multi-Tiered Pipelines: Apply lightweight compression at the edge, batch-level pruning at collectors, and serialized storage formats (e.g., Parquet, Protobuf) for archival use.

Designing such intelligent compression strategies would enable scalable and efficient provenance analysis without compromising fidelity or causal integrity.

7.3 Concluding Remarks

In conclusion, this thesis presents a unified, causality-preserving, and graph-centric methodology for forensic analysis and anomaly detection in distributed microservice-based systems. By incrementally building capabilities—from efficient log collection (*XPLOG*), to dynamic graph construction (μ*Prov*), and finally unsupervised anomaly detection using GNNs (μ*ProvGAE*)—this work addresses key challenges in distributed system observability, security, and scalability. The results presented herein demonstrate that lightweight and instrumentation-free mechanisms can still offer deep visibility and real-time insights into complex attack behaviors. While the systems presented lay a strong foundation, there remain exciting opportunities to enrich, scale, and operationalize these techniques further. With the growing reliance on distributed architectures, the ability to trace, explain, and mitigate sophisticated cyberattacks using dynamic provenance analysis will be increasingly critical for ensuring trust, resilience, and security in modern computing environments.

Bibliography

[1] S. Newman, *Monolith to microservices: evolutionary patterns to transform your monolith*. O'Reilly Media, 2019.

[2] S. Newman, *Building microservices: designing fine-grained systems*. O'Reilly Media, Inc., 2021.

[3] DIE.NET, "The linux audit daemon." https://linux.die.net/man/8/auditd, June 2024. [accessed 4. Feb. 2025].

[4] Linux, "syslog(3) — linux manual page." https://man7.org/linux/man-pages/man3/syslog.3.html, March 2007. Accessed: 2025-07-05.

[5] Linux, "journald.conf(5) — linux manual page." https://man7.org/linux/man-pages/man5/journald.conf.5.html, March 2007. Accessed: 2025-07-05.

[6] B. Li, X. Peng, Q. Xiang, H. Wang, T. Xie, J. Sun, and X. Liu, "Enjoy your observability: an industrial survey of microservice tracing and analysis," *Empirical Software Engineering*, vol. 27, pp. 1–28, 2022.

[7] C. Majors, L. Fong-Jones, and G. Miranda, *Observability Engineering*. O'Reilly Media, Inc., 2022.

[8] R. E. Kalman, "On the general theory of control systems," in *Proceedings first international conference on automatic control, Moscow, USSR*, pp. 481–492, 1960.

[9] W. U. Hassan, M. A. Noureddine, P. Datta, and A. Bates, "Omegalog: High-fidelity attack investigation via transparent multi-layer log analysis," in *The 27th Annual*

Network and Distributed System Security Symposium (NDSS 2020), San Diego, California, USA, 2020.

[10] J. Zeng *et al.*, "Watson: Abstracting behaviors from audit logs via aggregation of contextual semantics," in *The 28th Annual Network and Distributed System Security Symposium (NDSS 2021), Virtually*, 2021.

[11] A. Alshamrani, S. Myneni, A. Chowdhary, and D. Huang, "A survey on advanced persistent threats: Techniques, solutions, challenges, and research opportunities," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 2, pp. 1851–1877, 2019.

[12] X. Han, T. Pasquier, A. Bates, J. Mickens, and M. Seltzer, "Unicorn: Runtime provenance-based detector for advanced persistent threats," in *The 27th Annual Network and Distributed System Security Symposium (NDSS 2020), San Diego, California, USA*, 2020.

[13] F. Yang, J. Xu, C. Xiong, Z. Li, and K. Zhang, "Prographer: An anomaly detection system based on provenance graph embedding," in *The 32nd USENIX Security Symposium (USENIX Security 2023), Anaheim, CA, USA*, 2023.

[14] X. Ge, B. Niu, and W. Cui, "Reverse debugging of kernel failures in deployed systems," in *The USENIX Annual Technical Conference (USENIX ATC 2020), Boston, MA, USA*, 2020.

[15] Linux, "Linux fuse." https://man7.org/linux/man-pages/man4/fuse.4.html, 2001. [accessed 12. Jul. 2025].

[16] C. Schaufler, "Lsm: Stacking for major security modules." https://lwn.net/Articles/697259, 2016. [accessed 12. Jul. 2025].

[17] T. F. J.-M. Pasquier, J. Singh, D. Eyers, and J. Bacon, "Camflow: Managed data-sharing for cloud services," *IEEE Transactions on Cloud Computing (IEEE TCC 2017)*, vol. 5, no. 3, pp. 472–484, 2017.

[18] 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 Symposium (USENIX Security 2017), Vancouver, BC, Canada*, 2017.

[19] S. M. Milajerdi, B. Eshete, R. Gjomemo, and V. Venkatakrishnan, "Poirot: Aligning attack behavior with kernel audit records for cyber threat hunting," in *The ACM SIGSAC Conference on Computer and Communications Security (CCS 2019), London, UK*, 2019.

[20] S. M. Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. Venkatakrishnan, "Holmes: real-time apt detection through correlation of suspicious information flows," in *The IEEE Symposium on Security and Privacy (IEEE S&P 2019), San Francisco, CA, USA*, 2019.

[21] S. Ma, J. Zhai, F. Wang, K. H. Lee, X. Zhang, and D. Xu, "Mpi: Multiple perspective attack investigation with semantic aware execution partitioning," in *The 26th USENIX Security Symposium (USENIX Security 2017), Vancouver, BC, Canada*, 2017.

[22] K. H. Lee, X. Zhang, and D. Xu, "High accuracy attack provenance via binary-based execution partition.," in *The 20th Annual Network and Distributed System Security Symposium, (NDSS 2013), San Diego, California, USA*, 2013.

[23] K. H. Lee, X. Zhang, and D. Xu, "Loggc: Garbage collecting audit log," in *The ACM SIGSAC Conference on Computer and Communications Security (CCS 2013), Berlin, Germany*, 2013.

[24] S. Ma, X. Zhang, and D. Xu, "Protracer: Towards practical provenance tracing by alternating between logging and tainting.," in *The 23rd Annual Network and Distributed System Security Symposium (NDSS 2016), San Diego, California, USA*, 2016.

[25] L. Yu *et al.*, "ALchemist: Fusing application and audit logs for precise attack provenance without instrumentation," in *The 28th Annual Network and Distributed System Security Symposium, (NDSS 2021), Virtually*, 2021.

[26] R. Fonseca, G. Porter, R. H. Katz, and S. Shenker, "X-trace: A pervasive network tracing framework," in *The 4th Symposium on Networked Systems Design and Implementation (NSDI 2007), Cambridge, Massachusetts, USA*, 2007.

[27] J. Mace, R. Roelke, and R. Fonseca, "Pivot tracing: Dynamic causal monitoring for distributed systems," in *The 25th Symposium on Operating Systems Principles (SOSP 2015), Monterey, CA, USA*, 2015.

[28] Google, "Iopipe - monitor serverless applications." https://www.iopipe.com/, 2016. [accessed 12. Jul. 2025].

[29] Dashbird, "Dashbird - monitor serverless applications." https://dashbird.io/, 2017. [accessed 12. Jul. 2025].

[30] EPSAGON, "Epsagon - monitor serverless applications." https://epsagon.com/, 2017. [accessed 12. Jul. 2025].

[31] Google, "Thundra - monitor serverless applications." https://www.thundra.io/, 2018. [accessed 12. Jul. 2025].

[32] P. Datta, I. Polinsky, M. A. Inam, A. Bates, and W. Enck, "Alastor: Reconstructing the provenance of serverless intrusions," in *The 31st USENIX Security Symposium (USENIX Security 2022), Boston, MA, USA*, 2022.

[33] U. Satapathy, R. Thakur, S. Chattopadhyay, and S. Chakraborty, "Disprotrack: Distributed provenance tracking over serverless applications," in *The 42nd IEEE Conference on Computer Communications (INFOCOM 2023), New York City, NY, USA*, 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 (IEEE TDSC)*, vol. 21, no. 4, pp. 2085–2098, 2024.

[35] Datadog, "Datadog - log collection and integrations." https://docs.datadoghq.com/logs/log_collection/. [accessed 4. Feb. 2025].

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

[37] M. Bonola *et al.*, "Faster software packet processing on FPGA NICs with eBPF program warping," in *The USENIX Annual Technical Conference (USENIX ATC 2022), Carlsbad, CA, USA*, 2022.

[38] C. Liu, Z. Cai, B. Wang, Z. Tang, and J. Liu, "A protocol-independent container network observability analysis system based on eBPF," in *The 26th IEEE International Conference on Parallel and Distributed Systems (ICPADS 2020), Hong Kong*, 2020.

[39] Y. Zhong *et al.*, "Xrp: In-kernel storage functions with eBPF," in *The 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2022), Carlsbad, CA, USA*, 2022.

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

[41] A. Alsaheel, Y. Nan, S. Ma, L. Yu, G. Walkup, Z. B. Celik, X. Zhang, and D. Xu, "Atlas: A sequence-based learning approach for attack investigation," in *The 30th USENIX Security Symposium (USENIX Security 2021), Virtual*, 2021.

[42] P. Gao, X. Xiao, Z. Li, F. Xu, S. R. Kulkarni, and P. Mittal, "Aiql: Enabling efficient attack investigation from system monitoring data," in *The USENIX Annual Technical Conference (USENIX ATC 2018), Boston, MA, USA*, 2018.

[43] N. Yan, Y. Wen, L. Chen, Y. Wu, B. Zhang, Z. Wang, and D. Meng, "Deepro: Provenance-based APT campaigns detection via gnn," in *The 21st IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom 2022), Wuhan, China*, 2022.

[44] F. Zhang, R. Dai, and X. Ma, "Attrseq: Attack story reconstruction via sequence mining on causal graph," in *The 3rd IEEE International Conference on Power, Electronics and Computer Applications (ICPECA 2023)*, 2023.

[45] H. Liu and R. Jiang, "A causal graph-based approach for apt predictive analytics," *Electronics*, vol. 12, no. 8, p. 1849, 2023.

[46] M. M. Anjum, S. Iqbal, and B. Hamelin, "Anubis: a provenance graph-based framework for advanced persistent threat detection," in *The 37th ACM/SIGAPP Symposium on Applied Computing (SAC 2022), Virtual Event*, 2022.

[47] T. Pasquier, X. Han, M. Goldstein, T. Moyer, D. Eyers, M. Seltzer, and J. Bacon, "Practical whole-system provenance capture," in *The Symposium on Cloud Computing (SoCC 2017), Santa Clara, CA, USA*, 2017.

[48] A. Bates, D. J. Tian, K. R. Butler, and T. Moyer, "Trustworthy whole-system provenance for the linux kernel," in *24th USENIX Security Symposium (USENIX Security 2015), Washington, D.C., USA*, 2015.

[49] B. Bhattarai and H. H. Huang, "Prov2vec: Learning provenance graph representation for anomaly detection in computer systems," in *The 19th International Conference on Availability, Reliability and Security (ARES 2024), Vienna, Austria*, 2024.

[50] S. Wang, Z. Wang, T. Zhou, H. Sun, X. Yin, D. Han, H. Zhang, X. Shi, and J. Yang, "Threatrace: Detecting and tracing host-based threats in node level through provenance graph learning," *IEEE Transactions on Information Forensics and Security (IEEE TIFS)*, vol. 17, pp. 3972–3987, 2022.

[51] 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 *The 30th Annual Network and Distributed System Security Symposium (NDSS 2023), San Diego, California, USA*, 2023.

[52] A. Gehani and D. Tariq, "Spade: Support for provenance auditing in distributed environments," in *The ACM/IFIP/USENIX 13th International Middleware Conference (Middleware 2012), Montreal, QC, Canada*, 2012.

[53] J. Zhang, Y. Ren, and S. Kannan, "Fusionfs: Fusing i/o operations using ciscops in firmware file systems," in *20th USENIX Conference on File and Storage Technologies (FAST 2022), Santa Clara, CA, USA*, pp. 297–312, 2022.

[54] D. J. Pohly, S. McLaughlin, P. McDaniel, and K. Butler, "Hi-fi: collecting high-fidelity whole-system provenance," in *The 28th Annual Computer Security Applications Conference (ACSAC 2012), Orlando, FL, USA*, 2012.

[55] S. Ma, K. H. Lee, C. H. Kim, J. Rhee, X. Zhang, and D. Xu, "Accurate, low cost and instrumentation-free security audit logging for windows," in *The 31st Annual Computer Security Applications Conference (ACSAC 2015), Los Angeles, CA, USA*, 2015.

[56] Y. Xie, D. Feng, Y. Hu, Y. Li, S. Sample, and D. Long, "Pagoda: A hybrid approach to enable efficient real-time provenance based intrusion detection in big data envi-

ronments," *IEEE Transactions on Dependable and Secure Computing (IEEE TDSC)*, vol. 17, no. 6, pp. 1283–1296, 2018.

[57] S. M. Milajerdi, B. Eshete, R. Gjomemo, and V. Venkatakrishnan, "Poirot: Aligning attack behavior with kernel audit records for cyber threat hunting," in *The ACM SIGSAC Conference on Computer and Communications Security (CCS 2019), London, UK*, 2019.

[58] R. Yang, S. Ma, H. Xu, X. Zhang, and Y. Chen, "Uiscope: Accurate, instrumentation-free, and visible attack investigation for gui applications.," in *The 27th Annual Network and Distributed System Security Symposium (NDSS 2020) San Diego, California, USA*, 2020.

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

[60] J. Levin and T. A. Benson, "Viperprobe: Rethinking microservice observability with ebpf," in *The 9th IEEE International Conference on Cloud Networking (CloudNet 2020), Piscataway, NJ, USA*, 2020.

[61] F. Neves, N. Machado, *et al.*, "Falcon: A practical log-based analysis tool for distributed systems," in *The 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2018), Luxembourg City, Luxembourg*, 2018.

[62] T. Taylor, F. Araujo, and X. Shu, "Towards an open format for scalable system telemetry," in *The IEEE International Conference on Big Data (IEEE BigData 2020), Atlanta, GA, USA*, 2020.

[63] R. Sekar, H. Kimm, and R. Aich, "eaudit: A fast, scalable and deployable audit data collection system," in *The 45th IEEE Symposium on Security and Privacy (IEEE S&P 2024), San Francisco, CA, USA*, 2024.

[64] "Sysdig - the cloud moves fast. your security should too." https://sysdig.com/. [accessed 24. July. 2025].

[65] B. Debnath *et al.*, "Loglens: A real-time log analysis system," in *The 38th IEEE International Conference on Distributed Computing Systems (ICDCS 2018), Vienna, Austria*, 2018.

[66] K. Suo, Y. Zhao, W. Chen, and J. Rao, "vnettracer: Efficient and programmable packet tracing in virtualized networks," in *The 38th IEEE International Conference on Distributed Computing Systems (ICDCS 2018), Vienna, Austria*, 2018.

[67] K. Kc and X. Gu, "Elt: Efficient log-based troubleshooting system for cloud computing infrastructures," in *The 30th IEEE Symposium on Reliable Distributed Systems (SRDS 2011), Madrid, Spain*, 2011.

[68] 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 *The 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2014), Broomfield, CO, USA*, 2014.

[69] A. Ramachandran and M. Kantarcioglu, "Smartprovenance: A distributed, blockchain based dataprovenance system," in *The 8th ACM Conference on Data and Application Security and Privacy (CODASPY 2018), Tempe, AZ, USA*, 2018.

[70] M. Backes, S. Bugiel, and S. Gerling, "Scippa: System-centric ipc provenance on android," in *The 30th Annual Computer Security Applications Conference (ACSAC 2014), New Orleans, LA, USA*, 2014.

[71] 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.

[72] C. Collberg, A. Gibson, S. Martin, N. Shinde, A. Herzberg, and H. Shulman, "Provenance of exposure: Identifying sources of leaked documents," in *The 1st IEEE Conference on Communications and Network Security (CNS 2013), National Harbor, MD, USA*, 2013.

[73] 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 (IEEE TSC)*, vol. 12, no. 2, pp. 214–230, 2016.

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

[75] 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 (IEEE TSC)*, 2024.

[76] A. Security, "Why container observability matters." https://www.aquasec.com/cloud-native-academy/docker-container/container-monitoring/. [accessed 4. Feb. 2025].

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

[78] 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 *The 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2016), Savannah, GA, USA*, 2016.

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

[80] J. Lockerman *et al.*, "The fuzzylog: A partially ordered shared log," in *The 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2018), Carlsbad, CA, USA*, 2018.

[81] A. Ahmad, S. Lee, and M. Peinado, "Hardlog: Practical tamper-proof system auditing using a novel audit device," in *The 43rd IEEE Symposium on Security and Privacy (IEEE S&P 2022), San Francisco, CA, USA*, 2022.

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

[83] Y. Zhou, Z. Wang, S. Dharanipragada, and M. Yu, "Electrode: Accelerating distributed protocols with ebpf," in *The 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2023), Boston, MA*, 2023.

[84] 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 (IEEE TNSM)*, vol. 19, no. 3, pp. 2827–2838, 2022.

[85] P. Fang, P. Gao, C. Liu, E. Ayday, K. Jee, T. Wang, Y. F. Ye, Z. Liu, and X. Xiao, "Back-Propagating system dependency impact for attack investigation," in *The 31st USENIX Security Symposium (USENIX Security 2022), Boston, MA, USA*, 2022.

[86] L. Liang and S. Liu, "Event-triggered distributed attack detection and fault diagnosis," *IEEE Transactions on Instrumentation and Measurement (IEEE TIM 2022)*, vol. 72, pp. 1–11, 2022.

[87] S. LINUXAG, "Linux audit-subsystem design documentation for linux kernel 2.6, v0. 1," 2004. [accessed 12. Jul. 2025].

[88] M. Liu, Z. Xue, X. He, and J. Chen, "Scads: A scalable approach using spark in cloud for host-based intrusion detection system with system calls," *CoRR*, 2021.

[89] A. Goyal, J. Liu, A. Bates, and G. Wang, "Orchid: Streaming threat detection over versioned provenance graphs," *arXiv preprint arXiv:2408.13347*, 2024.

[90] H. Zhu and C. Gehrmann, "Kub-sec, an automatic kubernetes cluster apparmor profile generation engine," in *The 14th International Conference on COMmunication Systems & NETworkS (COMSNETS 2022), Bangalore, India*, 2022.

[91] Z. Li, Y. Wei, X. Shen, L. Wang, Y. Chen, H. Xu, S. Ji, F. Zhang, L. Hou, W. Liu, *et al.*, "Marlin: Knowledge-driven analysis of provenance graphs for efficient and robust detection of cyber attacks," *arXiv preprint arXiv:2403.12541*, 2024.

[92] W. U. Hassan, S. Guo, D. Li, Z. Chen, K. Jee, Z. Li, and A. Bates, "Nodoze: Combatting threat alert fatigue with automated provenance triage," in *The 26th Annual Network and Distributed System Security Symposium (NDSS 2019), San Diego, California, USA*, 2019.

[93] K. Kuusinen, V. Balakumar, S. C. Jepsen, S. H. Larsen, T. A. Lemqvist, A. Muric, A. Nielsen, and O. Vestergaard, "A large agile organization on its journey towards devops," in *The 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2018), Prague, Czech Republic*, 2018.

[94] K. Ojo-Gonzalez, R. Prosper-Heredia, L. Dominguez-Quintero, and M. Vargas-Lombardo, "A model devops framework for saas in the cloud," in *Advances and Applications in Computer Science, Electronics and Industrial Engineering*, pp. 37–51, 2021.

[95] F. Alder, N. Asokan, A. Kurnikov, A. Paverd, and M. Steiner, "S-faas: Trustworthy and accountable function-as-a-service using intel sgx," in *The 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS 2019), London, UK*, pp. 185–199, 2019.

[96] K. S.-P. Chang and S. J. Fink, "Visualizing serverless cloud application logs for program understanding," in *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2017), Raleigh, NC, USA*, 2017.

[97] J. Schleier-Smith, V. Sreekanti, A. Khandelwal, J. Carreira, N. J. Yadwadkar, R. A. Popa, J. E. Gonzalez, I. Stoica, and D. A. Patterson, "What serverless computing is and should become: The next phase of cloud computing," *Communications of the ACM*, pp. 76–84, 2021.

[98] D. Taibi, J. Spillner, and K. Wawruch, "Serverless computing-where are we now, and where are we heading?," *IEEE Software*, pp. 25–31, 2020.

[99] L. Gu, D. Zeng, J. Hu, H. Jin, S. Guo, and A. Y. Zomaya, "Exploring layered container structure for cost efficient microservice deployment," in *The 40th IEEE Conference on Computer Communications (INFOCOM 2021), Vancouver, BC, Canada*, 2021.

[100] Amazon, "Aws x-ray." https://aws.amazon.com/xray/, 2016. [accessed 12. Jul. 2025].

[101] Google, "Google cloud - viewing monitored metrics." https://cloud.google.com/functions/docs/monitoring/metrics/, 2008. [accessed 12. Jul. 2025].

[102] Microsoft, "Microsoft azure monitor." https://cloud.google.com/functions/docs/monitoring/metrics/, 2012. [accessed 12. Jul. 2025].

[103] S. Zawoad, R. Hasan, and K. Islam, "Secprov: Trustworthy and efficient provenance management in the cloud," in *The 37th IEEE Conference on Computer Communications (INFOCOM 2018), Honolulu, HI, USA*, 2018.

[104] W. U. Hassan, A. Bates, and D. Marino, "Tactical provenance analysis for endpoint detection and response systems," in *The IEEE Symposium on Security and Privacy (S&P 2020), San Francisco, CA, USA*, 2020.

[105] H. Irshad, G. Ciocarlie, A. Gehani, V. Yegneswaran, K. H. Lee, J. Patel, S. Jha, Y. Kwon, D. Xu, and X. Zhang, "Trace: Enterprise-wide provenance tracking for real-time apt detection," *IEEE Transactions on Information Forensics and Security (IEEE TIFS 2021)*, vol. 16, pp. 4363–4376, 2021.

[106] J. Tavori and H. Levy, "Tornadoes in the cloud: Worst-case attacks on distributed resources systems," in *The 40th IEEE Conference on Computer Communications (INFOCOM 2021), Vancouver, BC, Canada*, 2021.

[107] Q. Wang, W. U. Hassan, D. Li, K. Jee, X. Yu, K. Zou, J. Rhee, Z. Chen, W. Cheng, C. A. Gunter, *et al.*, "You are what you do: Hunting stealthy malware via data provenance analysis.," in *The 27th Annual Network and Distributed System Security Symposium (NDSS 2020) San Diego, California, USA*, 2020.

[108] P. Datta, I. Polinsky, M. A. Inam, A. Bates, and W. Enck, "Alastor: Reconstructing the provenance of serverless intrusions," in *31st USENIX Security Symposium (USENIX Security 2022), Boston, MA, USA*, 2022.

[109] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "Sok: (state of) the art of war: Offensive techniques in binary analysis," *The IEEE Symposium on Security and Privacy (IEEE S&P 2016), San Jose, CA, USA*, 2016.

[110] Zardus, "Angr." https://angr.io/, 2016. [accessed 12. Jul. 2025].

[111] 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.

[112] M. Scrocca, R. Tommasini, A. Margara, E. D. Valle, and S. Sakr, "The kaiju project: enabling event-driven observability," in *The 14th ACM International Conference on Distributed and Event-based Systems (DEBS 2020), Montreal, Quebec, Canada*, 2020.

[113] T. Blount, A. Chapman, M. Johnson, and B. Ludascher, "Observed vs. possible provenance (research track)," in *The 13th International Workshop on Theory and Practice of Provenance (TaPP 2021), Virtual*, 2021.

[114] 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.

[115] A. Chen, Y. Wu, A. Haeberlen, W. Zhou, and B. T. Loo, "Differential provenance: Better network diagnostics with reference events," in *The 14th ACM Workshop on Hot Topics in Networks (HotNets 2015) Philadelphia, PA, USA*, 2015.

[116] 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 *The 29th Annual Network and Distributed System Security Symposium, (NDSS 2022), San Diego, California, USA*, 2022.

[117] 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 *The 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2023), Vancouver, BC, Canada*, 2023.

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

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

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

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

[122] 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 Computing Surveys (CSUR)*, vol. 53, no. 1, pp. 1–36, 2020.

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

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

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

[126] Y. Gan *et al.*, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *The 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2019), Providence, RI, USA*, 2019.

[127] S. Boucher, A. Kalia, D. G. Andersen, and M. Kaminsky, "Putting the "micro" back in microservice," in *The USENIX Annual Technical Conference (USENIX ATC 2018), Boston, MA, USA*, 2018.

[128] S. Wang, Z. Ding, and C. Jiang, "Elastic scheduling for microservice applications in clouds," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 32, no. 1, pp. 98–115, 2020.

[129] A. Arora, S. Kulkarni, and M. Demirbas, "Resettable vector clocks," in *The 19th Annual ACM Symposium on Principles of Distributed Computing (ACM PODC 2000), Portland, Oregon, USA*, 2000.

[130] M. Raynal, "About logical clocks for distributed systems," *ACM SIGOPS Operating Systems Review (ACM SIGOPS OSR 1992)*, vol. 26, no. 1, pp. 41–48, 1992.

[131] V. Estivill-Castro and D. Wood, "A survey of adaptive sorting algorithms," *ACM Computing Surveys (CSUR)*, vol. 24, no. 4, pp. 441–476, 1992.

[132] Y. Tokusashi, H. T. Dang, F. Pedone, R. Soulé, and N. Zilberman, "The case for in-network computing on demand," in *The 14th ACM European Conference on Computer Systems (ACM EuroSys 2019), Dresden, Germany*, 2019.

[133] 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 (IEEE TGCN)*, vol. 5, no. 4, pp. 1722–1733, 2021.

[134] T. Cerny, M. J. Donahoo, and M. Trnka, "Contextual understanding of microservice architecture: current and future directions," *ACM SIGAPP Applied Computing Review*, vol. 17, no. 4, pp. 29–45, 2018.

[135] P. Agarwal and S. Moharir, "On exploiting edge resources for micro-service based saass," in *The 16th International Conference on COMmunication Systems & NETworkS (COMSNETS 2024), Bengaluru, India*, 2024.

[136] U. Zdun, P.-J. Queval, G. Simhandl, R. Scandariato, S. Chakravarty, M. Jelic, and A. Jovanovic, "Microservice security metrics for secure communication, identity management, and observability," *ACM Transactions on Software Engineering and Methodology (ACM TOSEM 2023)*, vol. 32, no. 1, pp. 1–34, 2023.

[137] A. Bambhore Tukaram, S. Schneider, N. E. Díaz Ferreyra, G. Simhandl, U. Zdun, and R. Scandariato, "Towards a security benchmark for the architectural design of microservice applications," in *Proceedings of the 17th International Conference on Availability, Reliability and Security (ACM ARES 2022), Vienna,Austria*, 2022.

[138] X. Chen *et al.*, "Clarion: Sound and clear provenance tracking for microservice deployments," in *The 30th USENIX Security Symposium (USENIX Security 2021), Virtual*, 2021.

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

[140] A. Ibrahim, S. Bozhinoski, and A. Pretschner, "Attack graph generation for microservice architecture," in *The 34th ACM/SIGAPP Symposium on Applied Computing (SAC 2019), Limassol, Cyprus*, 2019.

[141] D. Yuxin, Y. Xuebing, Z. Di, D. Li, and A. Zhanchao, "Feature representation and selection in malicious code detection methods based on static system calls," *Computers & Security*, vol. 30, no. 6-7, pp. 514–524, 2011.

[142] X. Xiao, Z. Wang, Q. Li, Q. Li, and Y. Jiang, "Anns on co-occurrence matrices for mobile malware detection," *KSII Transactions on Internet and Information Systems (TIIS 2015)*, vol. 9, no. 7, pp. 2736–2754, 2015.

[143] U. Satapathy, H. Borse, and S. Chakraborty, "Towards generating a robust, scalable and dynamic provenance graph for attack investigation over distributed microservice architecture," in *The 17th International Conference on COMmunication Systems and NETworks (COMSNETS 2025), Bengaluru, India*, 2025.

[144] N. DeMarinis, K. Williams-King, D. Jin, R. Fonseca, and V. P. Kemerlis, "Sysfilter: Automated system call filtering for commodity software," in *The 23rd International Symposium on Research in Attacks, Intrusions and Defenses (USENIX RAID 2020), San Sebastian, Spain*, 2020.

[145] M. Pancholi, A. D. Kellas, V. P. Kemerlis, and S. Sethumadhavan, "Timeloops: Automatic system call policy learning for containerized microservices," *arXiv preprint arXiv:2204.06131*, 2022.

[146] S. Yang, B. B. Kang, and J. Nam, "Optimus: association-based dynamic system call filtering for container attack surface reduction," *Journal of Cloud Computing*, vol. 13, no. 1, p. 71, 2024.

[147] K. P. Birman and T. A. Joseph, "Reliable communication in the presence of failures," *ACM Transactions on Computer Systems (ACM TOCS) 1987*, vol. 5, no. 1, pp. 47–76, 1987.

[148] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. I. Seltzer, "Provenance-aware storage systems," in *The USENIX Annual Technical Conference (USENIX ATC 2006), Boston, MA, USA*, 2006.

Thesis Related Publications

The following is a list of publications during my tenure as a PhD student in IIT Kharagpur. The publications are listed in chronological order.

Journals

1. **Utkalika Satapathy**, Harsh Borse, Rajat Bachhawat, Neha Dalmia, Subhrendu Chattopadhyay, and Sandip Chakraborty, "XPLOG: A Dynamic Observability Framework for Distributed Sandboxed Microservices", in IEEE Transactions on Services Computing (IEEE TSC), 2025. [Accepted]

2. **Utkalika Satapathy**, Harsh Borse and Sandip Chakraborty, "*μProvGAE*: Context-Enriched Provenance Graph for Attack Detection in Distributed Systems using Graph Autoencoders, in IEEE Transactions on Services Computing (IEEE TSC), 2025. [Submitted]

Conference Proceedings

1. **Utkalika Satapathy**, Rishabh Thakur, Subhrendu Chattopadhyay, Sandip Chakraborty, "Disprotrack: Distributed provenance tracking over serverless applications", in proc. of 42^{nd} IEEE Conference on Computer Communications (IEEE INFOCOM), 2023, New York area, USA

2. **Utkalika Satpathy**, Harsh Borse, Sandip Chakraborty, "Towards Generating a Robust, Scalable and Dynamic Provenance Graph for Attack Investigation over Distributed Microservice Architecture", in proc. of 17^{th} International Conference on Communication Systems & Networks (COMSNETS), 2025, Bangalore, India. (Received Best Paper Runner's Up Award)

Other Publications

The following is a list of other publications during my tenure as a student in IIT Kharagpur. The publications are listed in chronological order.

1. Neha Chowdhary, **Utkalika Satapathy**, Theophilus Benson, Subhrendu Chattopadhyay, Palani Kodeswaran, Sayandeep Sen, Sandip Chakraborty, "BeeGuard: Explainability-based Policy Enforcement of eBPF Codes for Cloud-native Environments", in proc. of 17^{th} International Conference on Communication Systems & Networks (COMSNETS), 2025, Bangalore, India.

2. Harsh Borse, **Utkalika Satapathy**, Mainack Mondal, Bivas Mitra, "URCD: Unsupervised Root Cause Detection in Microservices Architecture with HGAN", in proc. of 44th IEEE International Conference on Distributed Computing Systems (Poster) (IEEE ICDCS), 2024, New Jersey, USA.

3. Harsh Borse, **Utkalika Satapathy**, Mainack Mondal, Bivas Mitra, "SysResolve: Study on In-Context LLM Generation of Resolution Scripts", in proc. of 36th International Conference on Database and Expert Systems Applications (DEXA 2025), 2025, Bangkok, Thailand.

4. Harsh Borse, **Utkalika Satapathy**, Mainack Mondal, Bivas Mitra, "Microservices in Action: A Benchmark Dataset for Real-World Performance and Anomaly Analysis in Distributed Environments", In proc. of The 5th International Workshop on AI System Engineering: Math, Modelling and Software (AISys 2025), 2025, Bangkok, Thailand

5. Harsh Borse, **Utkalika Satapathy**, Mainack Mondal, Bivas Mitra, "MicroSuggest: Kernel-Aware Microservice Decomposition", in the proc. of The 27th International Conference on Big Data Analytics and Knowledge Discovery (DAWAK 2025), 2025, Bangkok, Thailand