Towards Generating a Robust, Scalable and Dynamic Provenance Graph for Attack Investigation over Distributed Microservice Architecture

Utkalika Satpathy, Harsh Borse, Sandip Chakraborty
Department of Computer Science and Engineering, IIT Kharagpur, India
Email: utkalika.satapathy01@kgpian.iitkgp.ac.in, harshzf2@gmail.com, sandipc@cse.iitkgp.ac.in

***Abstract*—In recent years, detecting sophisticated attacks in distributed microservice environments has become increasingly challenging, mainly due to containerization, which adds another dimension of complexity for collecting the system logs and the lack of applications designed with known vulnerabilities for reproducibility and experimentation. This paper presents a framework called *μProv* for generating robust, scalable, and dynamic provenance graphs to aid in attack investigation over distributed microservice architectures. Our approach captures fine-grained, system-level interactions across microservices leveraging eBPF and constructs dynamic runtime provenance graphs representing the causal relationships between processes, files, and network activities. We integrate real-world attack scenarios with known vulnerabilities into our system to evaluate its effectiveness. Through extensive empirical analysis, we demonstrate that *μProv* offers improved accuracy, scalability, and granularity compared to traditional logging methods.**

***Index Terms*—microservices; APTs; provenance; distributed**

I. Introduction

Modern large-scale distributed applications rely on microservice-based architecture [1], [2] 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 [3]–[5], enabling sophisticated attack vectors, such as *Advanced Persistent Threats* (APTs) [6], [7] 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 [7]. 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 [8]–[15], 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 [10], [12], [16], [17] 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.

(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.

(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.

(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.

Considering these challenges, we argue that designing a

[1]https://www.cve.org/ (Accessed: November 14, 2024)
[2]https://cwe.mitre.org/ (Accessed: November 14, 2024)

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 paper, 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 paper are as follows.

(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) [18], [19], 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 [20] 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.

(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.

(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.

(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.

II. Related Work

This section presents an overview of existing research and approaches in attack investigation, particularly over microservices, focusing on the detection and provenance of APT attacks. Among the existing works, `ATLAS` [21] introduced causality analysis on heterogeneous logs using NLP and ML to mitigate alert fatigue for analysts by constructing complete attack stories with high accuracy. Similarly, the authors in [22] designed a specialized *Attack Investigation Query Language* (AIQL) and data model to express key behaviors efficiently. As discussed, bridging system and application-level contexts is critical for holistic attack investigation. In this direction, `OmegaLog` [23] pioneered universal provenance to encode causal dependencies across layers through program control flow integrity. In a similar line, in [24], the authors have proposed an approach to tackle the complexity of large dependency graphs by assigning discriminative weights and propagating impacts to filter insignificant edges. In [25], the authors have developed detection schemes for cyber-physical multi-agent systems using event-triggered communication that reduces unnecessary communication and allows agents to achieve fault diagnosis independently, even if neighbor information is compromised. Advanced ML techniques have also gained traction for enhancing detection, pattern recognition, and reconstruction of security attacks over distributed applications. In this line, `Deepro` [9] employs graph neural networks on provenance graphs to identify APT campaigns based on data dependencies. `AttRSeq` [26] use attention-based LSTM models to learn attack patterns from causal relation sequences. In [27], the authors have combined causal graphs with deep learning for predictive APT analysis by modeling evolving malicious entity interactions. Similarly, `ANUBIS` [8] leverage Bayesian neural networks for high-fidelity predictions and explainable provenance of security attacks on microservice-based applications. Although these approaches have utilized a provenance-based approach for large-scale attack detection on microservices, however, they primarily look at individual microservices in silos or perform host-based system provenance analysis. In contrast, the real-time APT attacks trigger a chain of interactions across microservices, which needs runtime analysis of the whole system provenance rather than the provenance of individual microservices or hosts.

Application sandboxing is another significant challenge in microservice provenance, as discussed earlier. Traditional approaches, such as Linux Audit [28], offer limited visibility into sandboxed environments and struggle to capture critical sandbox-specific system events. Some prior research has focused on attack detection using only application-layer logs [29], while others, like [30], rely on analyzing system call traces from Linux applications in monolithic data centers. Recent works [7], [11], [12], [31]–[33] have highlighted the potential of data provenance, which offers more comprehensive context than raw system audit logs, for detecting host-based attacks. The enriched context provided by data provenance is beneficial for distinguishing persistent malicious activities from benign ones [34]. However, these techniques are not well-adapted for distributed microservice environments. State-of-the-art provenance frameworks such as [13], [14], [35], etc., are primarily designed to collect logs from applications running directly on the host kernel. However, due to sandboxing, an additional layer of abstraction complicates the extraction of system calls, as system events triggered within the sandbox environment are more challenging to capture.

III. 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) [36]. 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.

- $\mathbf{G_1}$: 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* [37] 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. 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.

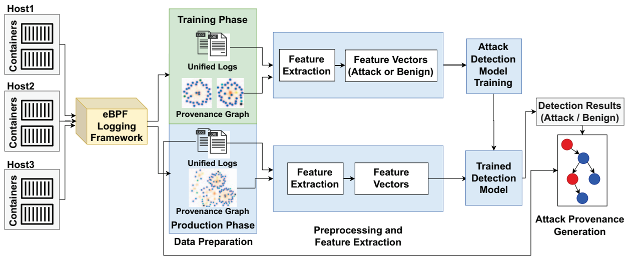


Fig. 1: *µProv*'s Architecture

IV. System Components

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

A. *eBPF-based Logging Framework*

We propose a logging framework leveraging eBPF [19] 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. 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. 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, and **(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 1 (discussed in the subsequent subsection).

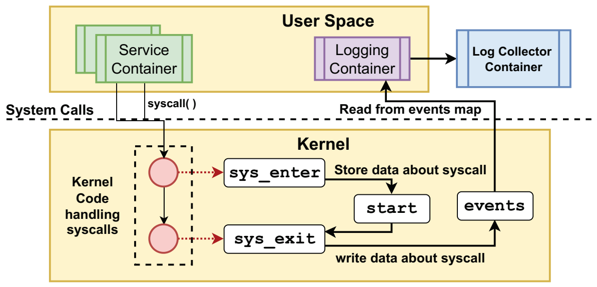


Fig. 2: eBPF-based Logging Framework

```
/* 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 1: Example of log entries generated using μ*Prov*

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 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 1). This is essential for separating log contexts across processes in various containers. In Fig. 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*" [38], 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.

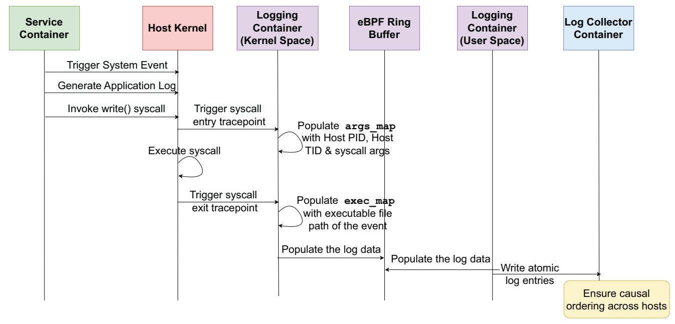

Service Container
Host Kernel
Logging Container (Kernel Space)
eBPF Ring Buffer
Logging Container (User Space)
Log Collector Container
Trigger System Event
Generate Application Log
Invoke write() syscall
Trigger syscall entry tracepoint
Populate args_map with Host PID, Host TID & syscall args
Execute syscall
Trigger syscall exit tracepoint
Populate exec_map with executable file path of the event
Populate the log data
Populate the log data
Write atomic log entries
Ensure causal ordering across hosts

Fig. 3: Atomic System Event Logging

B. *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 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] (❶ *File I/O related:* `read`, `write`, `open`, `close`, `dup`, `dup2`, `dup3`, `openat`, `unlinkat`; ❷ *Process related:* `clone`, `fork`, `vfork`, `execve`, `exit`, `exit_group`; and ❸ *Socket related:* `connect`, `accept`,

[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.

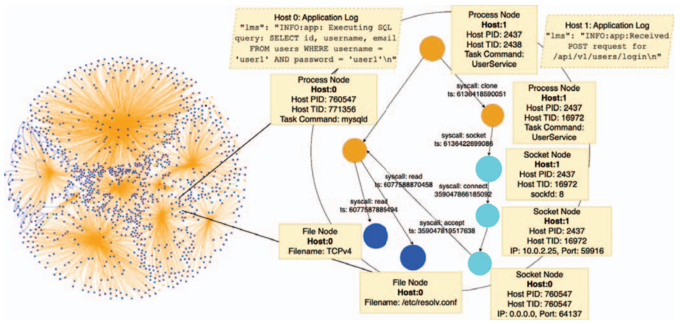

Host 0: Application Log
"lms": "INFO:app: Executing SQL query: SELECT id, username, email FROM users WHERE username = 'user1' AND password = 'user1'\n"
Process Node Host:1 Host PID: 2437 Host TID: 2438 Task Command: UserService
Host 1: Application Log
"lms": "INFO:app:Received POST request for /api/v1/users/login\n"
Process Node Host:0 Host PID: 760547 Host TID: 771356 Task Command: mysqld
syscall: clone ts: 6136418590051
Process Node Host:1 Host PID: 2437 Host TID: 16972 Task Command: UserService
syscall: socket ts: 6136422699086
Socket Node Host:1 Host PID: 2437 Host TID: 16972 sockfd: 8
syscall: read ts: 6077588870458
syscall: connect 359047866185092
syscall: read ts: 6077587889494
syscall: accept ts: 359047819517638
Socket Node Host:1 Host PID: 2437 Host TID: 16972 IP: 10.0.2.25, Port: 59916
File Node Host:0 Filename: TCPv4
File Node Host:0 Filename: /etc/resolv.conf
Socket Node Host:0 Host PID: 760547 Host TID: 760547 IP: 0.0.0.0, Port: 64137

Fig. 4: Provenance Graph generated from the global log file (Benign) for the *PicShare* application (Fig. 5). The magnified section shows the causal path across multiple hosts when a new user registers.

`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.

Incremental Graph Construction: Algorithm 1 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:

$$V_i = V_{i-1} \cup V_i' \quad (1)$$
$$E_i = E_{i-1} \cup E_i' \quad (2)$$
$$\mathcal{M}_i = \text{UpdateMappings}(\mathcal{M}_{i-1}, \mathcal{M}_i') \quad (3)$$

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:

$$f_s : (V \times G_{i-1} \times \mathcal{M}_{i-1} \times \tau) \to (V_i' \times E_i' \times \mathcal{M}_i') \quad (4)$$

Fig. 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.

V. 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) to emulate real-world vulnerabilities for distributed microservices.

Algorithm 1: Provenance Graph Generation

```
Procedure Main
    Input: $\mathcal{L}$: Global log file
    Output: $G = (V, E)$: Provenance graph
    // Initialization of variables
    $V \leftarrow \{v_{\text{exit}}, v_{\text{error}}\}$, $E \leftarrow \emptyset$;
    $\mathcal{M}_p, \mathcal{M}_f, \mathcal{M}_c, \mathcal{M}_n \leftarrow \emptyset$;
    /* $\mathcal{M}_p$: Process mapping (host_id, host_pid,
        host_tid) → Node                                  */
    /* $\mathcal{M}_f$: File descriptor mapping (host_id,
        host_pid, host_tid, fd) → Node                    */
    /* $\mathcal{M}_n$: Network Connection mapping (IP, port) →
        Node                                              */
    /* $\mathcal{M}_c$: Clone mapping (host_pid, retval, host_tid,
        type(parent or child)) → Node                     */
    foreach $l \in \mathbb{L}$ do
        /* $l$ is a syscall log entry with fields:
            host,pid,tid,ts,syscall,cgroup_id,...         */
        $(h, p, t, \tau, s, c, m, pns) \leftarrow$ ExtractInfo($l$);
        $v \leftarrow$ GCPN$h, p, t, c, m, pns, \mathcal{M}_p$;
        $(V', E', \mathcal{M}') \leftarrow$ 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($s, v, G, \mathcal{M}, \tau, l$)
    $V' \leftarrow \emptyset$, $E' \leftarrow \emptyset$, $\mathcal{M}' \leftarrow \emptyset$;
    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 $(V', E', \mathcal{M}')$

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 $(\mathcal{M}_p, \mathcal{M}_f, \mathcal{M}_c, \mathcal{M}_n)$
// Example of a specific syscall processing function
Function ProcessAccept4($v, l, G, \mathcal{M}, \tau$)
    $V' \leftarrow \emptyset$, $E' \leftarrow \emptyset$, $\mathcal{M}' \leftarrow \emptyset$;
    $fd \leftarrow l$.arguments.$fd$;
    $ip \leftarrow l$.artifacts.$IP$;
    $port \leftarrow l$.artifacts.$port$;
    if $(ip, port) \notin \mathcal{M}_n$ then
        $v_{net} \leftarrow$ CreateNetworkNode$(ip, port)$;
        $V' \leftarrow V' \cup \{v_{net}\}$;
        $\mathcal{M}'[(ip, port)] \leftarrow v_{net}$;
    $v_{net} \leftarrow \mathcal{M}_n[(ip, port)]$;
    $E' \leftarrow E' \cup \{(v, v_{net}, "accept4")\}$;
    $\mathcal{M}'[(v.pid, fd)] \leftarrow v_{net}$;
    return $(V', E', \mathcal{M}')$
```

TABLE I: 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

A. 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 I.

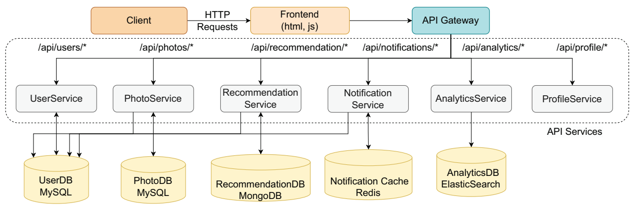


Fig. 5: The architecture of the `PicShare` Service for uploading, sharing, and viewing photos.

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, 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 I, 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.

B. 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. **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 V). 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. **Model Selection:** We explore standard supervised multiclass classification techniques commonly employed in traditional frameworks for detecting various benign and attack scenarios [10], [14]. Specifically, we investigate classification models such as K-Nearest Neighbors (KNN) [39], Support Vector Machines (SVM) [30], [39], Random Forest [7], and Artificial Neural Networks (ANN) [40], 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.

VI. Evaluation

We evaluate μ*Prov* 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 μ*Prov* in comparison to Tracee.

A. 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)*, *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 with known vulnerabilities (refer to Table I). 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 nodes.

[4]https://github.com/aquasecurity/tracee (Accessed: November 14, 2024)
[5]https://anonymous.4open.science/r/MicroserviceProv-76CE/README.md

B. Results

1) 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 μ*Prov* as shown in Table II, Random Forest shows the highest performance in μ*Prov*, 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 μ*Prov* consistently outperforms Tracee in every metric due to its causally-ordered comprehensive logging information.

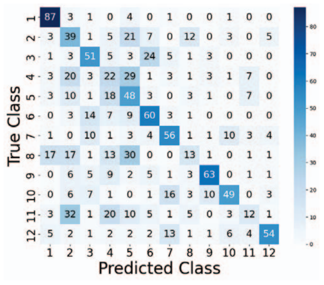
TABLE II: Performance Comparison for our Framework

Model Name	**Tracee**				μ***Prov***			
	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

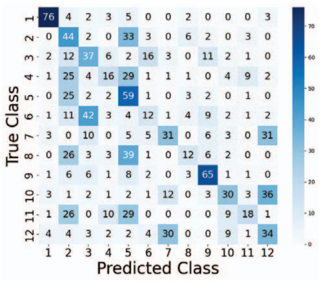
Fig. 7 and Fig. 8 show the confusion matrix of different machine learning models (KNN, SVM, Random Forest, and ANN) using Tracee and μ*Prov* in identifying attack scenarios across 12 classes of 'benign' and 'attack' activities. Across all the models, μ*Prov* consistently outperforms Tracee in classifying attack and benign events. This highlights the effectiveness of μ*Prov*'s causally-ordered logs, which provide a richer context for distinguishing. Also, it shows Random Forest exhibits the best overall performance for Tracee and μ*Prov*, 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 μ*Prov* 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. 7 and Fig. 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, μ*Prov* 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, μ*Prov* reduces the number of FP (benign

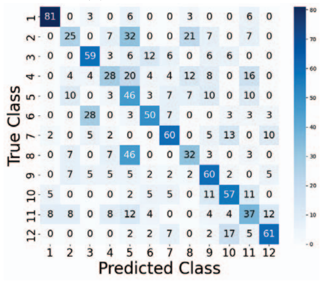
1 - Download Photo **2** - Get Profile **3** - Get Recommendation **4** - Login User **5** - Register User **6** - Update Recommendation
7 - View Photo **8** - Node EJS Attack - Profile **9** - RCE Attack - Photo **10** - SQL Injection Attack - Photo **11** - SQL Injection Attack - Users **12** - Upload PHP Attack - Photo



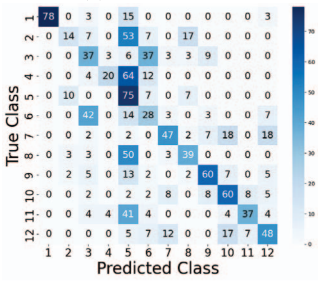
(a) Tracee - KNN



(b) Tracee - SVM

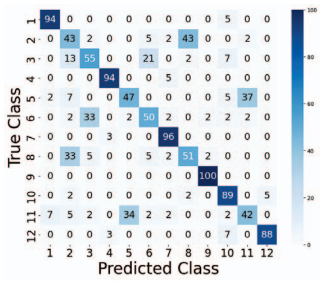


(c) Tracee - Random Forest

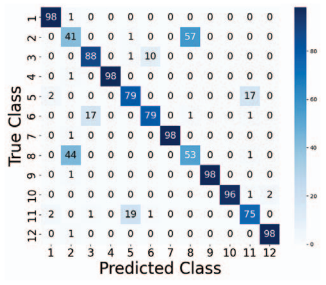


(d) Tracee - ANN

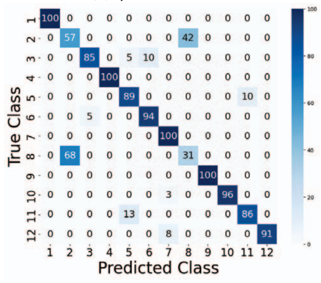
Fig. 7: Confusion Matrix for Tracee (%)



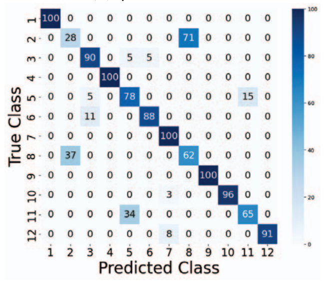
(a) μ*Prov*- KNN



(b) μ*Prov*- SVM



(c) μ*Prov*- Random Forest



(d) μ*Prov*- ANN

Fig. 8: Confusion Matrix for μ*Prov* (%)

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.

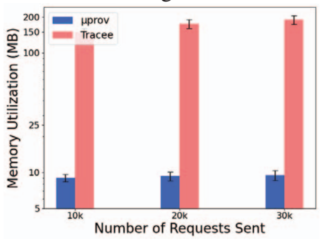
We compare the attack detection time between μ*Prov* and Tracee. For Tracee, the process includes generating logs across hosts, pre-processing, storing them centrally, generating a provenance graph, and running the pre-trained detection model. μ*Prov*, however, streams logs directly to a central server, eliminating the need for relevant log retrieval. We observe that on average, μ*Prov* detects attacks in 5.2 ± 0.5 seconds while Tracee takes 10.2 ± 0.6 seconds from the attack occurrence, making μ*Prov* $\sim 50\%$ faster.

2) Resource Utilization: To evaluate the resource overhead, we have used an open-source monitoring tool developed by Google to monitor container, *cAdvisor*[6]. As shown in Fig. 9a, the memory overhead of the logging framework of μ*Prov* remains stable across 10K, 20K, and 30K requests, around 10MB, whereas Tracee takes around 200MB. Fig. 9b shows the CPU utilization overhead by μ*Prov* and Tracee. The results shown in Fig. 10 describe the properties of the generated provenance graph for μ*Prov* and Tracee across 12 different events. As shown, μ*Prov* 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.

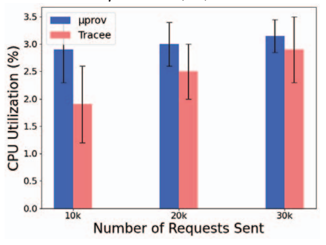
VII. Conclusion

This paper introduces μ*Prov*, an application-agnostic framework to capture fine-grained system interactions and construct comprehensive provenance graphs, enabling robust tracking of causal relationships among system entities that help detect real-time attacks for distributed microservice architecture. Unlike traditional system call tracing tools used for sandboxing applications, which often overlook key data, μ*Prov* captures comprehensive execution details that significantly enhance the ability to trace causal relationships among system entities. As a future extension, we plan to develop a framework that further integrates detailed system interactions involving both execution events and performance counters that capture the resource consumption by individual microservices, thus improving the detection of advanced attacks.

[6]https://github.com/google/cadvisor (Accessed: November 14, 2024)

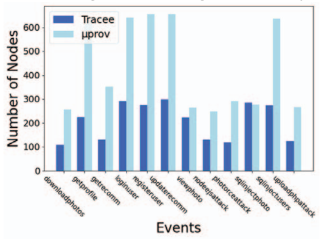


(a) Avg. Memory Usage

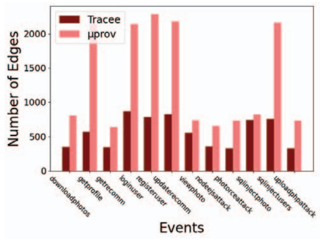


(b) Avg. CPU Utilization

Fig. 9: Average Memory overhead and CPU utilization



(a) No of Nodes Generated per Event



(b) No of Edges Generated per Event

Fig. 10: Properties of Provenance Graph

References

[1] 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.

[2] P. Agarwal and S. Moharir, “On exploiting edge resources for microservice based SaaSs,” in *2024 16th International Conference on COMmunication Systems & NETworkS (COMSNETS)*. IEEE, 2024, pp. 533–541.

[3] 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*, vol. 32, no. 1, pp. 1–34, 2023.

[4] 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*, 2022, pp. 1–7.

[5] X. Chen, H. Irshad, Y. Chen, A. Gehani, and V. Yegneswaran, “CLARION: Sound and clear provenance tracking for microservice deployments,” in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 3989–4006.

[6] 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.

[7] X. Han, T. Pasquier, A. Bates, J. Mickens, and M. Seltzer, “UNICORN: Runtime provenance-based detector for advanced persistent threats,” in *NDSS Symposium*, 2020.

[8] M. M. Anjum, S. Iqbal, and B. Hamelin, “ANUBIS: a provenance graph-based framework for advanced persistent threat detection,” in *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, 2022, pp. 1684–1693.

[9] N. Yan, Y. Wen, L. Chen, Y. Wu, B. Zhang, Z. Wang, and D. Meng, “Deepro: Provenance-based APT campaigns detection via GNN,” in *2022 IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, 2022, pp. 747–758.

[10] 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*, vol. 17, pp. 3972–3987, 2022.

[11] A. Goyal, J. Liu, A. Bates, and G. Wang, “ORCHID: Streaming threat detection over versioned provenance graphs,” *arXiv preprint arXiv:2408.13347*, 2024.

[12] 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 *NDSS*, 2020.

[13] A. Gehani and D. Tariq, “SPADE: Support for provenance auditing in distributed environments,” in *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 2012, pp. 101–120.

[14] T. Pasquier, X. Han, M. Goldstein, T. Moyer, D. Eyers, M. Seltzer, and J. Bacon, “Practical whole-system provenance capture,” in *Proceedings of the 2017 Symposium on Cloud Computing*, 2017, pp. 405–418.

[15] 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.

[16] B. Bhattarai and H. H. Huang, “Prov2vec: Learning provenance graph representation for anomaly detection in computer systems,” in *Proceedings of the 19th International Conference on Availability, Reliability and Security*, 2024, pp. 1–14.

[17] 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 *30th Network and Distributed System Security Symposium*, 2023.

[18] C. Liu, Z. Cai, B. Wang, Z. Tang, and J. Liu, “A protocol-independent container network observability analysis system based on eBPF,” in *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2020, pp. 697–702.

[19] S. Miano, F. Risso, M. V. Bernal, M. Bertrone, and Y. Lu, “A framework for eBPF-based network functions in an era of microservices,” *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 133–151, 2021.

[20] J. Levin and T. A. Benson, “ViperProbe: Rethinking microservice observability with ebpf,” in *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*. IEEE, 2020, pp. 1–8.

[21] 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 *30th USENIX security symposium (USENIX Security)*, 2021, pp. 3005–3022.

[22] P. Gao, X. Xiao, Z. Li, F. Xu, S. R. Kulkarni, and P. Mittal, “AIQL: Enabling efficient attack investigation from system monitoring data,” in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 113–126.

[23] W. U. Hassan, M. A. Noureddine, P. Datta, and A. Bates, “OmegaLog: High-fidelity attack investigation via transparent multi-layer log analysis,” in *Network and Distributed System Security Symposium (NDSS)*, 2020.

[24] 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 *31st USENIX Security Symposium (USENIX Security)*, 2022, pp. 2461–2478.

[25] L. Liang and S. Liu, “Event-triggered distributed attack detection and fault diagnosis,” *IEEE Transactions on Instrumentation and Measurement*, vol. 72, pp. 1–11, 2022.

[26] F. Zhang, R. Dai, and X. Ma, “AttRSeq: Attack story reconstruction via sequence mining on causal graph,” in *2023 IEEE 3rd International Conference on Power, Electronics and Computer Applications (ICPECA)*. IEEE, 2023, pp. 360–364.

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

[28] S. LINUXAG, “Linux audit-subsystem design documentation for linux kernel 2.6, v0. 1,” 2004.

[29] R. Bronte, H. Shahriar, and H. M. Haddad, “Mitigating distributed denial of service attacks at the application layer,” in *Proceedings of the ACM Symposium on Applied Computing (ACM SAC)*, 2017, p. 693–696.

[30] 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,” *arXiv preprint arXiv:2109.11821*, 2021.

[31] S. M. Milajerdi, B. Eshete, R. Gjomemo, and V. Venkatakrishnan, “Poirot: Aligning attack behavior with kernel audit records for cyber threat hunting,” in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (ACM CCS)*, 2019, pp. 1795–1812.

[32] S. M. Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. Venkatakrishnan, “Holmes: real-time apt detection through correlation of suspicious information flows,” in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1137–1152.

[33] H. Zhu and C. Gehrmann, “Kub-Sec, an automatic kubernetes cluster apparmor profile generation engine,” in *2022 14th International Conference on COMmunication Systems & NETworkS (COMSNETS)*, 2022, pp. 129–137.

[34] 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 *network and distributed systems security symposium*, 2019.

[35] A. Bates, D. J. Tian, K. R. Butler, and T. Moyer, “Trustworthy Whole-System provenance for the linux kernel,” in *24th USENIX Security*, 2015, pp. 319–334.

[36] A. Ibrahim, S. Bozhinoski, and A. Pretschner, “Attack graph generation for microservice architecture,” in *Proceedings of the 34th ACM/SIGAPP symposium on applied computing*, 2019, pp. 1235–1242.

[37] F. Neves, N. Machado, R. Vilaça, and J. Pereira, “Horus: Non-intrusive causal analysis of distributed systems logs,” in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2021, pp. 212–223.

[38] A. Arora, S. Kulkarni, and M. Demirbas, “Resettable vector clocks,” in *Proceedings of the 19th annual ACM Symposium on Principles of Distributed Computing (ACM PODC)*, 2000, pp. 269–278.

[39] 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.

[40] 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)*, vol. 9, no. 7, pp. 2736–2754, 2015.