DisProTrack: Distributed Provenance Tracking over Serverless Applications

Utkalika Satapathy*, Rishabh Thakur*, Subhrendu Chattopadhyay†, Sandip Chakraborty*
*IIT Kharagpur, Kharagpur, India 721302 †IDRBT, Hyderabad, India 500057

***Abstract*—Provenance tracking has been widely used in the recent literature to debug system vulnerabilities and find the root causes behind faults, errors, or crashes over a running system. However, the existing approaches primarily developed graph-based models for provenance tracking over monolithic applications running directly over the operating system kernel. In contrast, the modern DevOps-based service-oriented architecture relies on distributed platforms, like serverless computing that uses container-based sandboxing over the kernel. Provenance tracking over such a distributed micro-service architecture is challenging, as the application and system logs are generated asynchronously and follow heterogeneous nomenclature and logging formats. This paper develops a novel approach to combining system and micro-services logs together to generate a Universal Provenance Graph (UPG) that can be used for provenance tracking over serverless architecture. We develop a 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. Finally, we design a regular expression-based log optimization method for reverse query parsing over the generated UPG. A thorough evaluation of the proposed UPG model with different benchmarked serverless applications shows the system's effectiveness.**

***Index Terms*—Distributed provenance tracking,**

I. Introduction

Modern service-oriented architecture adopts DevOps [1], [2] practices and technologies to provide Software as a service (SaaS) [3] by leveraging distributed cloud infrastructure. Service deployment on top of the cloud widely adopts serverless computing (SLC) [4] 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 [4]–[7]. However, the available serverless-specific industry solutions [8]–[10] 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 [11]–[14] support distributed tracing as well as cost analysis features by instrumenting the source codes. However, these tools only support applications developed using a particular programming language. So, it is difficult to analyze the actual behavior of these micro-services.

Provenance Graphs: The non-invasive [1] frameworks rely on the logs generated by applications to identify the execution states. Since most of the production-grade micro-services 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 [15]–[20]. 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 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 [21], [22]. 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.

A. 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 [16]–[20], [23]. 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 micro-service over an SLC application are physically distributed across the entire eco-system, 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 micro-services can provide a meaningful platform for distributed provenance tracking. A few existing works [20], [24] have tried to address the issue of encoding all forensically-relevant causal dependencies regardless of their origin. Nevertheless, we have some non-trivial

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

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 straight-forward, particularly when a micro-service run 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 micro-services are deployed in different sandboxed environments.

• Challenge (4) – *Dependency explosion and handling confounding root causes:* To find out the root cause behind an event, the system administrator needs to execute a reverse query on the UPG. The results obtained from the reverse query can be multiple due to partial matching of the input query string. This results in more than one root cause behind a query event. Further, due to plausible circular dependencies among the micro-services, the resultant UPG might not be a DAG. Therefore, it is non-trivial to track all the causal paths behind an event.

Owing to the above challenges, in this paper, 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 application's container logs. The core idea behind `DisProTrack` is to judicially use the applications' control flow graph (CFG) to avoid the runtime log tracking complexities.

B. Our Contributions

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

1. Design of the 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.

2. Runtime execution unit identification: We develop a Linux LKM (loadable kernel module) which 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 which 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.

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.

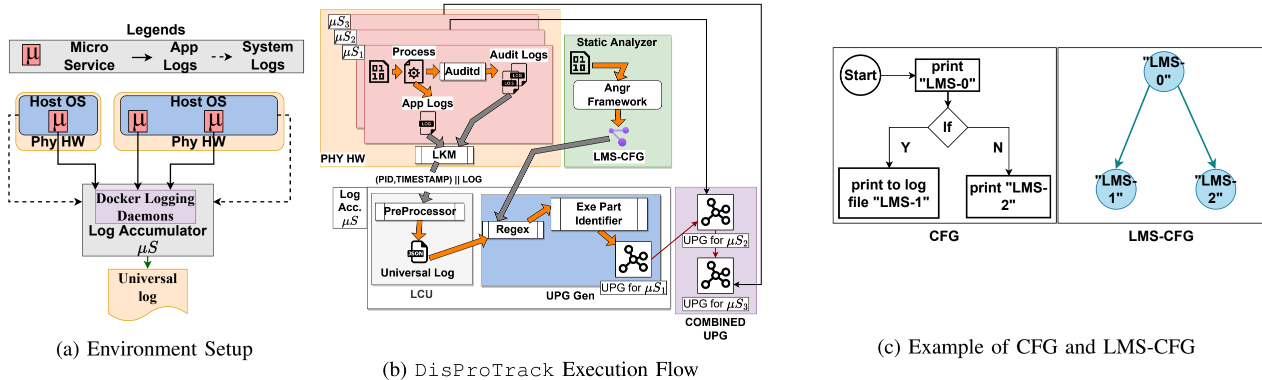
4. Implementation and evaluation: We have implemented the proposed framework, which can be deployed as a micro-service on top of the SLC without instrumenting the source code of the applications. We have made the implementation open-sourced[2]. Based on the experimental evaluation of `DisProTrack` with 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.

II. Related Work

Existing third party log collection tools like, `FUSE` [25], `LSM` [26], `CamFlow` [27], `SPADE` [28], etc., allow hooking the kernel level objects and system calls; however, these methods require instrumentation of the OS. Additionally, the collection of system-level provenance in a containerized or serverless environment is difficult due to the micro-services' distributed and minimalistic deployed nature. Therefore, the existing threat detection and investigations with system-level provenance graph using kernel audit-log data, such as, `ETW` [29], `SLEUTH` [30], `Pagoda` [31], `POIROT` [32], `HOLMES` [33], `WATSON` [34], etc., do not suit an SLC environment. One common challenge for causality analysis with system provenance graph is the *"Dependency Explosion"* problem [35], [36] where too many *"root causes"* are behind one suspicious event. This dependency explosion increases the probability of *"security alert fatigue"* and *"missing threats"*.

`BEEP` [36] and `OMEGALOG` [16] have addressed dependency explosion problem by increasing input and output edge identification accuracy in the provenance graph. Other methods

[2]https://anonymous.4open.science/r/Project_ALV_2022-CEFD/ (Accessed: January 13, 2023)



(a) Environment Setup

(b) `DisProTrack` Execution Flow

(c) Example of CFG and LMS-CFG

Fig. 1: `DisProTrack` Overview

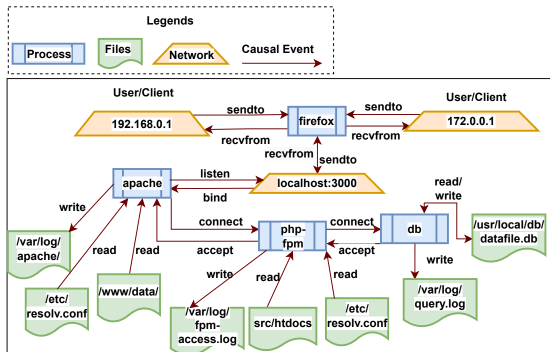


Fig. 2: An Example of System Provenance

like `ETW` [29], `LogGC` [37], `MPI` [35], `ProTracer` [38], etc., used execution partitioning to reduce the dependency explosion. However, most of these works require some instrumentation over the application source code. On the other hand, `ALCHEMIST` [20] and `OMEGALOG` used a combination of application-specific logs with system-level logs to track the information flow more accurately. In the same line, `UIScope` [39] proposed an instrumentation free, execution partition-based causality analysis for attack investigation system. However, the existing works are targeted toward monolithic systems and can not be deployed to secure SLC applications. Although, a few existing tools like, `X-trace` [40] and `PivotTracing` [41] are used to instrument SLC applications, they do not provide non-invasive property. Similarly, [11]–[14] are reliant on the used programming languages of the micro-service applications; thus lack flexibility [24]. The absence of language-independent, non-invasive causality analysis frameworks for SLC has motivated us to design `DisProTrack`.

III. DisProTrack Overview

The proposed `DisProTrack` is an open source provenance tracking system for containerized SLC as shown in Fig. 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. 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 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. 1b.

A. *`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 a LMS as a string present in the executable responsible for printing some log message. Typical LMSes contains format specifiers, error codes, debug level identifiers, etc. Since we only require the causal relationships between LMSes, the static analyzer identifies only the *Log Message Generating Functions* (LMGF).

Definition 1 (Log Message Generating Functions)**.** We define a LMGF as a library function that is directly used for printing a LMSes 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 IV-A.

Definition 2 (Log Message String CFG)**.** A LMS-CFG is formally defined as a directed graph $G' = (V', E')$ where $\forall_{e'_{i,j} \in E'} e'_{i,j} = (v'_i, v'_j) : v'_i, v'_j \in V'$ 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,

$\underset{e' \in E'}{\exists} e' = (v'_i, v'_k) : v'_i, v'_j, v'_k \in V', \underset{v'_j}{\nexists} v'_j \in \mathfrak{P}(G, v'_i, v'_k)$. 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. 1c where the constructed LMS-CFG contains only the LMS nodes from the CFG.

1) 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 IV-B2).

B. `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` [42] 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.

1) 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 IV-B2).

2) 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 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. 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 Section IV-C).

IV. 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*.

A. *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 LMSes with syscalls in between. In our proposed framework, we load the executable application binary and use the python `Angr` module [43], [44] to identify the CFG from it. The generated CFG is a directed graph having the basic instruction blocks (syscalls, printing of LMSes, 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 LMGFs 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 a LMGF name, then the algorithm extracts the arguments of the LMGF, which has been defined as a 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 V-B1). For accurate identification of the LMSes, 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

```
1  Procedure Main
     Input: G(V, E): CFG, 𝕃: Set of LMGFs in G(V, E)
     Output: LMS-CFG G' = (V', E')
2    Initialization:
3      V' ← ∅, E' ← ∅
4    for v ∈ 𝕃 do
       /* The Angr tools return whether a node
          is a loop                                  */
5      if v has a loop then
         /* For a loop, we need to find out
            all the LMSes (𝔸) that are printed
            through the loop.                        */
6        𝔸 ← BackTraceOptimization(BT_th, G(V, E), v);
         /* Construct the subgraph G_A(V_A, E_A)
            for A                                    */
7        G_A(V_A, E_A) ← CreateSubGraph(A_i);
8        V' ← V' ∪ V_A;
9        E' ← E' ∪ E_A;
10   for v' ∈ V' do
11     Identify format specifiers in v' and replace them with suitable
       Regular Expressions;
12   return G'(V', E');
13 Function BackTraceOptimization
     Input: BT: Backtrace Threshold, G(V, E): CFG, v: A node from 𝕃
            having a loop
     Output: 𝔸: A set of LMSes in the loop
14   Initialization:
15     𝔸 ← ∅
16   if v has a set of LMSes {l_0, ··· l_k} printed through the loop with
     syscalls in between the LMSes then
       /* We consider the loops having a
          syscall and associated LMSes               */
17     𝔸 ← 𝔸 ∪ {l_0, ··· , l_k};
18     return 𝔸;
19   else
       /* The loop within v does not print any
          LMSes                                      */
20     do
         /* Backtrack to the LMGF that called
            v, until the backtrack threshold
            BT is reached.                           */
21       if ⟨v̄ → v⟩ is a directed edge in G(V, E) then
22         if v̄ has a loop with a syscall and a set of LMSes
           {l_0, ··· l_k} then
23           𝔸 ← 𝔸 ∪ {l_0, ··· , l_k};
24           return 𝔸;
25       v ← v̄;
26       BT ← BT − 1;
27     while BT > 0;
28   return 𝔸;
29 Function CreateSubGraph
     Input: 𝔸
     Output: G_A(V_A, E_A): A Subgraph generated from 𝔸
30   Initialization:
31     V_A ← ∅, E_A ← ∅
32   foreach {l_i, l_{i+1}} ∈ 𝔸 do
33     V_A ← V_A ∪ {l_i, l_{i+1}};
34     E_A ← E_A ∪ {⟨l_i → l_{i+1}⟩};
35   return G_A(V_A, E_A);
```

B. 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.

1) 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 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.

2) 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 ($\langle pid, ts, lms\rangle$) and Syslog ($\langle pid, ts, syscall, path, exe\rangle$) entries. Using Algorithm 2, we identify the execution units in the ASCL file with

[3] https://docs.docker.com/storage/storagedriver/ (Accessed: January 13, 2023)

Algorithm 2: UPG Construction

```
1  Procedure Main
     Input: G' = (V', E'): LMS-CFG, F_pid: ASCL file for process pid
     Output: G^U_pid = (V^U_pid, E^U_pid): UPG component for process pid
2    Initialization:
3      V^U_pid ← ∅, E^U_pid ← ∅, U_pid ← ∅, E_pid ← ∅, G_pid ← ∅;
       /* U_pid: An execution unit denoting the
          set of LMSes matched with V'                 */
       /* E_pid: Set of system logs corresponding
          to an execution unit                         */
       /* G_pid: Set of all E_pid from F_pid           */
4      end_unit ← false /* tracks execution units      */
5    foreach e in F_pid do
       /* e can be an application-level entry
          ⟨pid, ts, lms⟩ or a syslog entry
          ⟨pid, ts, syscall, path, exe⟩                */
       /* 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                  */
6      if e is an application-level entry then
7        if e is the first entry in 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.             */
8          n ← FindLMSinGraph(G', e);
9          U_pid ← U_pid ∪ {n}
10       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                            */
11         n ← MatchNeighbourNodes(G', n, e);
12         U_pid ← U_pid ∪ n ;
13       if n is a leaf node in G'(V', E') then
14         end_unit ← true /* Seen a block of
              LMSes logged by the application
              from a LMGF                              */
15     if end_unit is true then
16       E_pid ← E_pid ∪ e;
17       end_unit ← false /* tracked syslogs for an
            execution unit                             */
18       G_pid ← G_pid ∪ E_pid;
19       E_pid ← ∅
20     else
         /* e is a syslog entry and end_unit is
            false                                      */
21       E_pid ← E_pid ∪ e;
     /* Tracked all syslogs for individual
        execution units, now construct the UPG         */
22   partition ← 0 /* keep tracks of individual
        execution units                                */
23   foreach E_pid ∈ G_pid do
24     partition ← partition + 1 ;
25     foreach e ∈ E_pid do
26       if e has a valid exe and syscall then
27         V^U_pid ← V^U_pid ∪ {e.exe} /* the name of
              the executable application
              binary becomes a node                    */
28         V^U_pid ← V^U_pid ∪ {partition||e.path} /* the
              partition value appended with the
              object path accessed in e
              becomes the second node                  */
29         E^U_pid ← E^U_pid ∪ {⟨e.exe →
              partition||e.path, e.syscall⟩} /* an
              edge is added between the above
              two nodes with e.syscall as the
              edge label                               */
```

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 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 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.

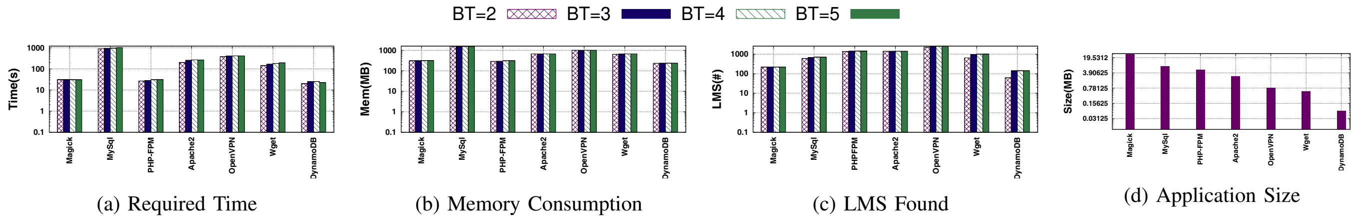
V. Performance Evaluation

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.

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

[4] https://anonymous.4open.science/r/Project_ALV_2022-CEFD/



(a) Required Time (b) Memory Consumption (c) LMS Found (d) Application Size

Fig. 3: Static Analysis – The Y-axes are in logarithmic scale

A. 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 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 IV-B1. On the other hand, the proposed static analyzer and runtime engine are deployed as a containerized micro-service.

B. 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, and the mean is shown in the plots.

1) Overhead of Static analyzer: To understand the resource utilization overhead during static analysis, we have considered 7 different applications as given in Table I.

TABLE I: 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

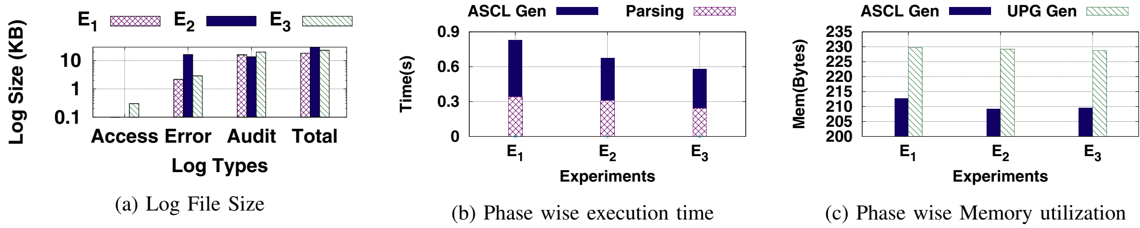
Based on the obtained results (Figs. 3a and 3b), we observe that the static analysis for `MySql` takes the longest time to complete and needs a greater amount of memory. The time and memory requirements increase when the BT increases. This is justified as the value of BT increases, the number of backtraces also increases (as mentioned in Section IV-A). However, an increase in the number of backtraces can identify more number of LMSes as shown in Fig. 3c. We also observe that even though the size of `Magick` is greater than the size of `MySql` (Fig. 3d), 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 application. 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.

2) 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. 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 popup. 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. 4a. 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

[5]https://www.docker.com/ (Accessed: January 13, 2023)
[6]https://docs.docker.com/engine/swarm/ (Accessed: January 13, 2023)
[7]https://hub.docker.com/ (Accessed: January 13, 2023)

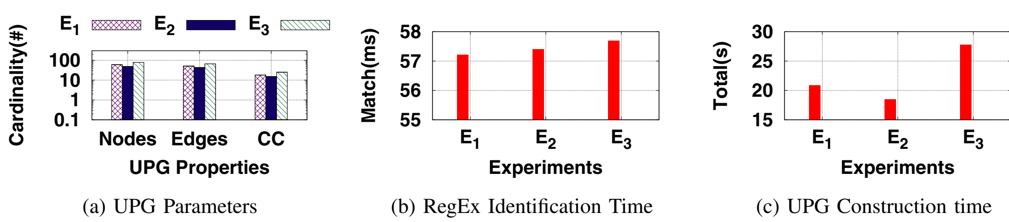


(a) Log File Size

(b) Phase wise execution time

(c) Phase wise Memory utilization

Fig. 4: Runtime Analysis – The Y-axis in (a) is in logarithmic scale



(a) UPG Parameters

(b) RegEx Identification Time

(c) UPG Construction time

Fig. 5: Runtime Analysis – The Y-axis in (a) is in logarithmic scale

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. 4b. 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. 5c) 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. 5b). We also provide the memory consumption during the individual phases as presented in Fig. 4c 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. 5a. 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.

VI. Analysis

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.

A. Adversarial model & Attack Scenario

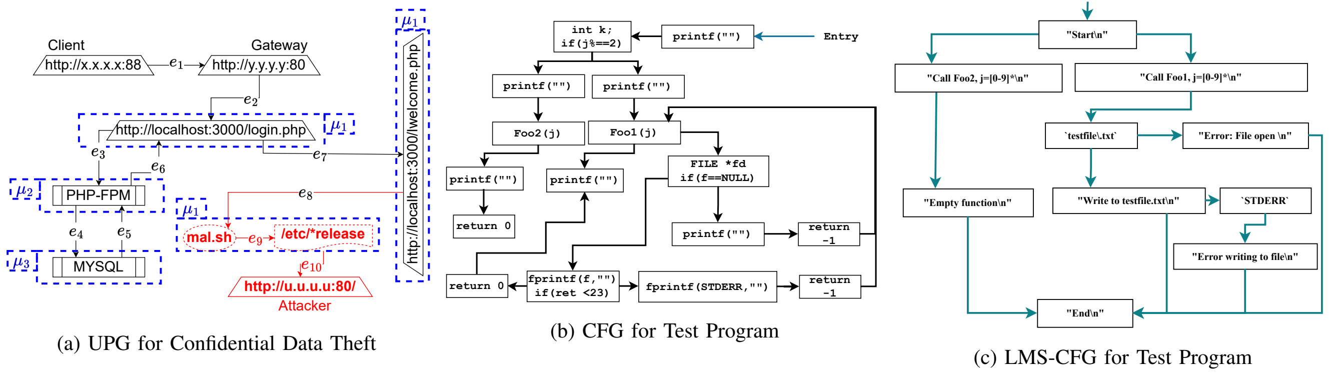
We assume that an adversary can somehow bypass the authorization mechanisms and gain access to one/more application container(s) except the runtime engine container without being detected. In the compromised container(s), the adversary can add, modify, and/or execute scripts and deploy webpages. However, 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] 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 V-B2 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.

B. 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. 6a. To ensure

[8]https://bit.ly/trendmicro_shading_light (Accessed: January 13, 2023)



(a) UPG for Confidential Data Theft

(b) CFG for Test Program

(c) LMS-CFG for Test Program

Fig. 6: A PoC Case Study to Analyze the Accuracy of `DisProTrack`

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.).

```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 1: PoC Program for Accuracy Analysis

C. 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. 3c 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 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. 6c. For comparison purposes, we present the corresponding CFG in Fig. 6b, 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.

VII. Conclusion

This paper developed a non-invasive causality analysis framework, called `DisProTrack`, for provenance tracking over distributed serverless applications. The proposed framework is capable of adversarial attack analysis by identifying the root causes effectively. `DisProTrack` can be deployed on top of the SLC as a micro-service and has the virtue of being lightweight and provides results within 0.5 minutes. A PoC analysis of `DisProTrack` also shows its efficiency and efficacy in detecting attack instances for an SLC application.

A critical aspect of `DisProTrack` is that it uses a heuristic to identify the execution units by matching the CFGs generated from the micro-service binaries with the runtime application and system logs based on their temporal execution patterns. Thus, the framework might wrongly identify the execution units if the underlying servers running the micro-services are not weakly time synchronized (time drift within a small threshold). Nevertheless, this condition rarely occurs in a typical distributed SLC platform with multiple micro-services interacting with each other. Further, `DisProTrack` can be deployed as an additional micro-service along with the other application micro-services running over the servers, making it robust to be applied for a wide range of production-grade serverless application scenarios.

The authors have provided public access to their code and/or data at https://github.com/usatpath01/DisProTrack.

References

[1] 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 *Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2018)*, 2018, pp. 60–63.

[2] 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*, 2021, pp. 37–51.

[3] F. Alder, N. Asokan, A. Kurnikov, A. Paverd, and M. Steiner, "S-faas: Trustworthy and accountable function-as-a-service using intel sgx," in *ACM SIGSAC Conference on Cloud Computing Security Workshop*, 2019, pp. 185–199.

[4] 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)*, 2017, pp. 261–265.

[5] 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.

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

[7] L. Gu, D. Zeng, J. Hu, H. Jin, S. Guo, and A. Y. Zomaya, "Exploring layered container structure for cost efficient microservice deployment," in *IEEE Conference on Computer Communications (IEEE INFOCOM 2021)*, 2021, pp. 1–9.

[8] "Aws x-ray," https://aws.amazon.com/xray/.

[9] "Google cloud - viewing monitored metrics," https://cloud.google.com/functions/docs/monitoring/metrics/.

[10] "Microsoft azure monitor," https://cloud.google.com/functions/docs/monitoring/metrics/.

[11] "Iopipe - monitor serverless applications," https://www.iopipe.com/.

[12] "Dashbird - monitor serverless applications," https://dashbird.io/.

[13] "Thundra - monitor serverless applications," https://www.thundra.io/.

[14] "Epsagon - monitor serverless applications," https://epsagon.com/.

[15] S. Zawoad, R. Hasan, and K. Islam, "Secprov: Trustworthy and efficient provenance management in the cloud," in *IEEE Conference on Computer Communications (IEEE INFOCOM 2018)*, 2018, pp. 1241–1249.

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

[17] W. U. Hassan, A. Bates, and D. Marino, "Tactical provenance analysis for endpoint detection and response systems," in *IEEE Symposium on Security and Privacy (SP 2020)*. IEEE, 2020, pp. 1172–1189.

[18] M. M. Anjum, S. Iqbal, and B. Hamelin, "ANUBIS: a provenance graph-based framework for advanced persistent threat detection," in *ACM/SIGAPP Symposium on Applied Computing*, 2022, pp. 1684–1693.

[19] 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*, vol. 16, pp. 4363–4376, 2021.

[20] L. Yu, S. Ma, Z. Zhang, G. Tao, X. Zhang, D. Xu, V. E. Urias, H. W. Lin, G. F. Ciocarlie, V. Yegneswaran *et al.*, "ALchemist: Fusing application and audit logs for precise attack provenance without instrumentation." in *The Network and Distributed System Security Symposium (NDSS 2021)*, 2021.

[21] A. Bates, D. J. Tian, K. R. Butler, and T. Moyer, "Trustworthy {Whole-System} provenance for the linux kernel," in *USENIX Security Symposium (USENIX Security 2015)*, 2015, pp. 319–334.

[22] J. Tavori and H. Levy, "Tornadoes in the cloud: Worst-case attacks on distributed resources systems," in *IEEE Conference on Computer Communications (IEEE INFOCOM 2021)*, 2021, pp. 1–10.

[23] 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 Network and Distributed System Security Symposium (NDSS 2020)*, 2020.

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

[25] "Linux fuse," https://man7.org/linux/man-pages/man4/fuse.4.html.

[26] C. Schaufler, "Lsm: Stacking for major security modules," https://lwn.net/Articles/697259, 2016.

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

[28] A. Gehani and D. Tariq, "Spade: Support for prmiscovenance auditing in distributed environments," in *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 2012, pp. 101–120.

[29] 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 *Annual Computer Security Applications Conference (ACSAC 2015)*, 2015, pp. 401–410.

[30] 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 *USENIX Security Symposium (USENIX Security 2017)*, 2017, pp. 487–504.

[31] 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 environments," *IEEE Transactions on Dependable and Secure Computing*, vol. 17, no. 6, pp. 1283–1296, 2018.

[32] S. M. Milajerdi, B. Eshete, R. Gjomemo, and V. Venkatakrishnan, "Poirot: Aligning attack behavior with kernel audit records for cyber threat hunting," in *ACM SIGSAC Conference on Computer and Communications Security*, 2019, p. 1795–1812.

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

[34] J. Zeng, Z. L. Chua, Y. Chen, K. Ji, Z. Liang, and J. Mao, "Watson: Abstracting behaviors from audit logs via aggregation of contextual semantics," in *The Network and Distributed System Security Symposium (NDSS 2021)*, 2021.

[35] 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 *USENIX Security Symposium (USENIX Security 2017)*, 2017, pp. 1111–1128.

[36] K. H. Lee, X. Zhang, and D. Xu, "High accuracy attack provenance via binary-based execution partition." in *The Network and Distributed System Security Symposium (NDSS 2013)*, vol. 2, 2013, p. 4.

[37] ——, "Loggc: Garbage collecting audit log," in *ACM conference on Computer & communications security (SIGSAC 2013)*, 2013, pp. 1005–1016.

[38] S. Ma, X. Zhang, and D. Xu, "Protracer: Towards practical provenance tracing by alternating between logging and tainting." in *The Network and Distributed System Security Symposium (NDSS 2016)*, vol. 2, 2016, p. 4.

[39] R. Yang, S. Ma, H. Xu, X. Zhang, and Y. Chen, "UIScope: Accurate, instrumentation-free, and visible attack investigation for gui applications." in *The Network and Distributed System Security Symposium (NDSS 2020)*, 2020.

[40] R. Fonseca, G. Porter, R. H. Katz, and S. Shenker, "X-Trace: A pervasive network tracing framework," in *USENIX Symposium on Networked Systems Design & Implementation (NSDI 2007)*, 2007.

[41] J. Mace, R. Roelke, and R. Fonseca, "Pivot tracing: Dynamic causal monitoring for distributed systems," in *Symposium on Operating Systems Principles (SOSP 2015)*, 2015, pp. 378–393.

[42] (2007, Mar) auditd(8) - linux manual page. [Online]. Available: https://man7.org/linux/man-pages/man8/auditd.8.html

[43] 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," *IEEE Symposium on Security and Privacy (SP 2016)*, 2016.

[44] "Angr," https://angr.io/.