Towards Scalable Cluster Auditing through Grammatical Inference over Provenance Graphs

Wajih Ul Hassan, Mark Lemay,‡ Nuraini Aguse, Adam Bates, Thomas Moyer*

University of Illinois at Urbana-Champaign
{whassan3,aguse2,batesa}@illinois.edu

‡ *Boston University*
lemay@bu.edu

* *UNC Charlotte*
tom.moyer@uncc.edu

***Abstract*—Investigating the nature of system intrusions in large distributed systems remains a notoriously difficult challenge. While monitoring tools (e.g., Firewalls, IDS) provide preliminary alerts through easy-to-use administrative interfaces, attack reconstruction still requires that administrators sift through gigabytes of system audit logs stored locally on hundreds of machines. At present, two fundamental obstacles prevent synergy between system-layer auditing and modern cluster monitoring tools: 1) the sheer volume of audit data generated in a data center is prohibitively costly to transmit to a central node, and 2) system-layer auditing poses a "needle-in-a-haystack" problem, such that hundreds of employee hours may be required to diagnose a single intrusion.**

This paper presents Winnower, a scalable system for audit-based cluster monitoring that addresses these challenges. Our key insight is that, for tasks that are replicated across nodes in a distributed application, a model can be defined over audit logs to succinctly summarize the behavior of many nodes, thus eliminating the need to transmit redundant audit records to a central monitoring node. Specifically, Winnower parses audit records into provenance graphs that describe the actions of individual nodes, then performs grammatical inference over individual graphs using a novel adaptation of Deterministic Finite Automata (DFA) Learning to produce a behavioral model of many nodes at once. This provenance model can be efficiently transmitted to a central node and used to identify anomalous events in the cluster. We have implemented Winnower for Docker Swarm container clusters and evaluate our system against real-world applications and attacks. We show that Winnower dramatically reduces storage and network overhead associated with aggregating system audit logs, by as much as 98%, without sacrificing the important information needed for attack investigation. Winnower thus represents a significant step forward for security monitoring in distributed systems.

I. Introduction

When investigating system intrusions, auditing large compute clusters remains a costly and error-prone process. Security monitoring tools such as firewalls and antivirus provide an efficient preliminary alert system for administrators, quickly notifying them if a suspicious activity such as a malware signature or a blacklisted IP is spotted somewhere in the cluster. However, determining the veracity and context of these compromise indicators still ultimately requires the inspection of system-layer audit logs. Unfortunately, auditing systems are not scaling to meet the needs of modern computing paradigms. System logs generate gigabytes of information per node per day, making it impractical to proactively store and process these records centrally. Moreover, the volume of audit information transforms attack reconstruction into a "needle-in-a-haystack" problem. In Advanced Persistent Threat (APT) scenarios, this reality delays incident response for months [48] as security teams spend hundreds to thousands of employee hours stitching together log records from dozens of machines.

The audit problem is only further exacerbated by the growing popularity of container-based virtualization, which has enabled rapid deployment and extreme scalability in datacenters and other multi-tenant environments [22]. Containers represent the realization of the microservice architecture principle [59], a popular pattern that encourages applications to run as discrete, loosely-coupled, and replicated services to provide scalability and fault-tolerance. However, the rapid adoption of containers has outpaced system administrators' ability to apply control and governance to their production environments. Container marketplaces such as Docker Store [4] now host over 0.5 million containers and boast over 8 billion downloads [1]; while these services simplify the sharing of applications, they also create a new ecosystem in which poorly maintained or malicious code is permitted to spread. These containers have no security guarantees and can contain vulnerabilities that could be used as attack vectors [65], [63]. Recently, Red Hat surveyed enterprises to figure out technical factors which prevent the use of containers in production and 75% of enterprises claimed security to be a major concern [9].

Data provenance, metadata that describes the lineage of data transformed by a system, is a promising new approach to system auditing. In the context of operating systems, provenance-based techniques parse kernel-layer audit records into a causal graph that describes the history of system execution [20], [58], [50], [47], [60]. The applications for data provenance are numerous, ranging from database management [30], [35], [45], networks diagnosis and debugging [26], [27], and forensic reconstruction of a chain of events after an attack [18], [72], [19], [67]. Unfortunately, even state-of-the-art provenance-based techniques are not presently applicable to the cluster auditing problem as they suffer from huge storage

Network and Distributed Systems Security (NDSS) Symposium 2018
18-21 February 2018, San Diego, CA, USA
ISBN 1-891562-49-5
http://dx.doi.org/10.14722/ndss.2018.23141
www.ndss-symposium.org

overhead and "needle-in-a-haystack" problem. While work on reducing provenance storage overhead exists [54], [51], [69], [25]; these systems lack scalability required for auditing large clusters.

In this paper, we present Winnower, a system that leverages provenance graphs as the basis for *online* modeling the behavior of applications that have been replicated across different nodes in a cluster. Winnower provides a storage- and network-efficient means of transmitting audit data to a central node for cluster-wide monitoring. The output of Winnower is a *provenance model* that concisely describes the behavior of hundreds of nodes, and can be used by system administrators to identify abnormal behaviors in the cluster. *Our key insight is that, because cluster applications are replicated in accordance with microservice architecture principle, the provenance graphs of these instances are operationally equivalent (i.e., highly redundant) except in the presence of anomalous activity.* Thus, recognition and removal of equivalent activity from provenance graphs will simultaneously solve both challenges associated with cluster auditing.

At the core of Winnower is a novel adaptation of *graph grammar* techniques. Inspired by formal grammars for string languages, graph grammars provide rule-based mechanisms for generating, manipulating and analyzing graphs [71], [44]. We demonstrate how graph grammar models can be learned over system-level provenance graphs through use of Deterministic Finite Automata (DFA) learning, a restrictive class of graph grammars which encodes the causality in generated models. These models can be used to determine whether new audit events are already described by the model, or whether the model needs to be incrementally updated. This approach made possible a series of *graph abstraction* techniques that enable DFA learning to generalize over the provenance of multiple nodes despite the presence of instance-specific information such as hostnames and process IDs. Combining these two features, Winnower can transmit and store causal information to the central monitoring node in a cost-effective manner and generate concise provenance graphs without sacrificing the information needed to identify attacks.

This paper makes the following contributions:

- To motivate our use of the container ecosystem as an exemplar, we conduct an analysis of Docker Store that uncovers high severity vulnerabilities, justifying the need for auditing tools (§II);
- We design a novel adaptation of graph grammars that demonstrates their applicability for system auditing. While to the best of our knowledge, this is the first use of grammatical inference over data provenance, we foresee additional security applications in the areas of information flow monitoring and control (§III);
- We present the Winnower, a proof-of-concept implementation that enables cluster auditing for Docker Swarm, Docker's cluster management tool (§IV). Winnower augments the Linux Audit System (`auditd`) to make it container-aware, providing a means for fine-grained provenance of container-based applications. In the evaluation, we demonstrate that Winnower reduces the overheads of cluster auditing by as much as 98% (§V).
- To determine the efficacy of Winnower for cluster auditing, we undertake an expansive series of case studies. Across five real-world attack scenarios, we demonstrate that Winnower dramatically simplifies attack reconstruction as compared to traditional methods based on `auditd` (§VI).

II. Background & Motivation

A. Docker Ecosystem

Docker is the most widely used container-based technology [1] which allows users to create self-contained applications with all dependencies built-in the form of *images*. Docker Store is an online registry that allows Docker users to share application container images online. Docker images are built on top of other images; for example, we can create LAMP stack image by selecting an Ubuntu distribution as a *base image*, then add Apache, MySQL, and PHP images. Currently, Docker Store contains two types of public image sharing repositories: *official* repositories that contain curated application containers verified by the vendors and *community* repositories that contain application containers from the public. At the time of this study, there are 140 official containers on Docker Store.

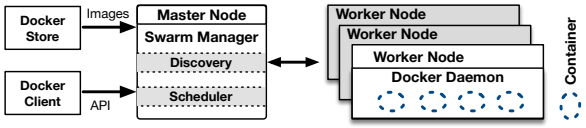
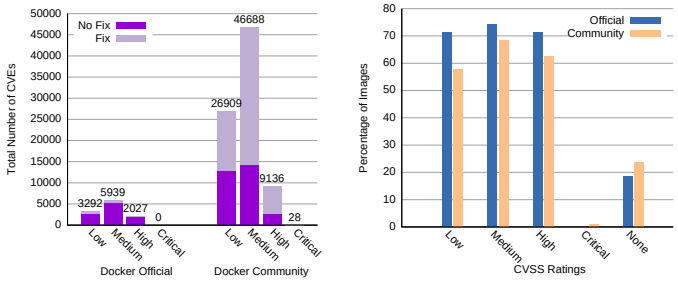


Fig. 1: Docker Swarm Architecture.

In this work, we used Docker Swarm which is the native Docker cluster management tool for resource allocation and dynamic scheduling of containers. However, our system is extensible to other cluster management tools (see §VIII). The basic architecture of Docker Swarm cluster is shown in Figure 1. Docker Swarm manager (master) is a frontend for users to control containers in the cluster while actual container executes on Docker Swarm nodes (workers). Docker Swarm users can specify `replication` factor when launching containers. Replication is useful to provide isolation, horizontal auto-scaling, load-balancing and fault-tolerance to services. For example, the following command will deploy 12 Nginx containers in Docker Swarm cluster.

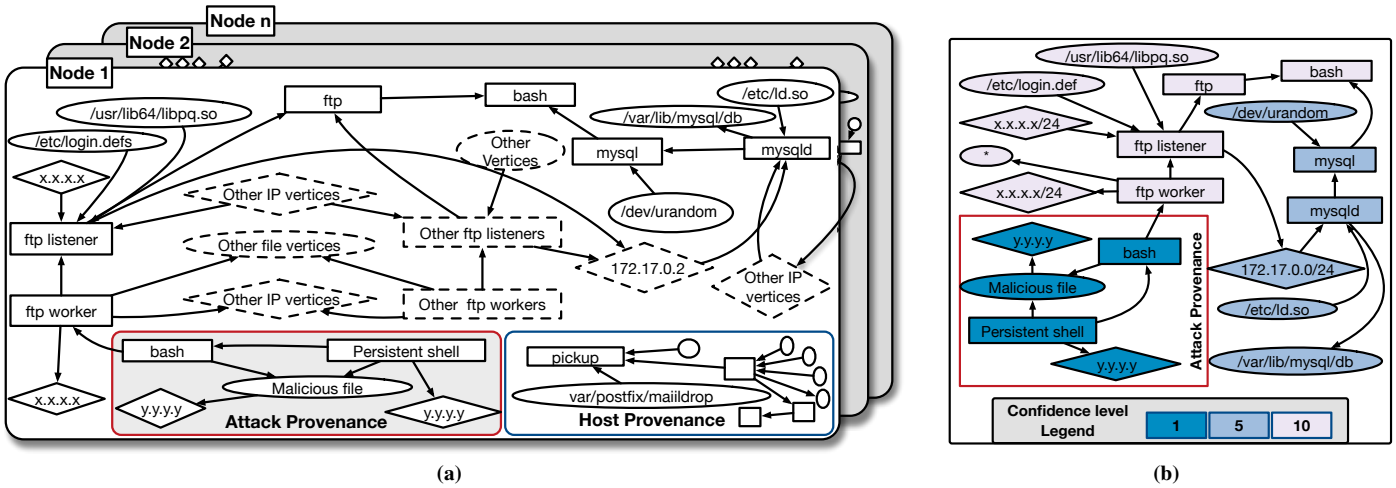
```
docker service create --name nginx --replicas 12 nginx
```



(a) Results of CVE Scan by severity for all 140 Docker Official images and the top 500 most popular Docker Community images.

(b) Percentage of Docker images that have at least one specified severity CVE in them

Fig. 2: Docker Store Security Analysis Results.



(a) (b)

Fig. 3: (a) Part of raw provenance graphs generated on master node using `auditd`. **(b)** Concise Provenance Graph generated by Winnower for monitoring cluster-wide behaviour with confidence levels. Diamonds, circles and rectangles shows socket (artifact), files (artifact), and process (activity) respectively. We removed edge relationships for readability.

Security Analysis of Docker Store. To demonstrate the potential security risks in the container ecosystem, we downloaded 140 official images and top 500 community images from Docker Store. Then, using Anchore's `cve-scan` tool [7] we statically analyzed the operating system packages built-in or downloaded by these Docker images. `cve-scan` categorizes the vulnerabilities present in container images using the Common Vulnerability Scoring System V3 (CVSS)[1] provided by CVE database, which specifies four severity levels for vulnerabilities: Low, Medium, High, and Critical. As shown in Figure 2a and 2b, `cve-scan` uncovers thousands of CVE's in both official and community images.[2] Moreover, we found that over 70% of official images have at least one High severity vulnerability (Figure 2b). These results indicate that security threats abound in the container ecosystem, underscoring the importance of developing runtime auditing solutions to container clusters.

B. Motivating Attack Scenario

To characterize the limitations of existing cluster auditing systems, we now consider a concrete scenario in which audit records can be used to diagnose an attack – an online file storage webservice which allows users to upload and download files using FTP. The webservice consists of a cluster with one master node and 4 worker nodes running 10 ProFTPD-1.3.3c containers configured with multiple worker daemons backed with 5 MySQL database containers for authentication. ProFTPD-1.3.3c version is vulnerable to a *remote code execution* attack[3]. We configured the worker nodes to stream complete descriptions of their activities to the master node using the Linux audit subsystem (`auditd`) [5], a widely used forensic tracking tool for Linux [70]. While the master node aggregated audit records from the workers' `auditd` streams, we generated a 3 minute workflow of heterogeneous requests during which an attack was launched on one of the nodes' container. The attack used the ProFTPD-1.3.3c vulnerability to obtain `bash` access and download a backdoor program to gain persistent access to the container.

The master node's view of the cluster is shown in Figure 3a, where the worker nodes' `auditd` streams are represented as provenance directed acyclic graph (DAG) [39] detailing the causal relations of the system. The graph has been simplified for readability; each node's graph was roughly 2 MB in size and contained around 2,000 vertices. The subgraph titled **Attack Provenance** only appears in Node 1, whereas the remainder of Node 1's graph is operationally equivalent to the activities of the other nodes. Based on this exercise, we observe fundamental limitations to leveraging system-layer audits in large clusters:

– *Graph Complexity.* Ideally, an administrator would have the tools necessary to quickly pinpoint an attack and identify the affected resources, but unfortunately the inspection of system-layer audit logs in a large cluster poses a needle-in-a-haystack problem. While we drew attention to the attack subgraph in Figure 3a, in practice this exercise can be extraordinarily tedious and error-prone [47], [38]. As demonstrated here, provenance graph visualization can assist in forensic exploration [23], [56], but such techniques are designed for a single-host and thus lack the means to filter the inherent redundancy across nodes.
– *Storage Overhead.* The amount of audit data generated on even a single host can be enormous, around 3.18GB/-day(server) and 1.2GB/day (client), as shown by previous studies [51], [36], [47]. When considering that such records may need to be stored in perpetuity for *post-facto* causal analysis, it immediately becomes clear that audit logs represent a storage burden on the order of terabytes. While prior work has made inroads at reducing storage burden for single hosts [54], [51], [69], [25], even state-of-the-art systems lack the scalability required for auditing large clusters.
– *Network Overhead.* Beyond the cost of local storage, cluster auditing requires aggregation of system activity to a central master node. However, it is immediately apparent that a naïve approach that transmits all system-layer audit records

[1] See https://nvd.nist.gov/vuln-metrics/cvss

[2] `cve-scan` does not analyze vulnerabilities in package managers such as NPM, PIP, and Maven. Nor does it detect insecure configuration settings, making our assessment a conservative lower bound on the severity of Docker insecurity.

[3] Available at https://www.exploit-db.com/exploits/15662/

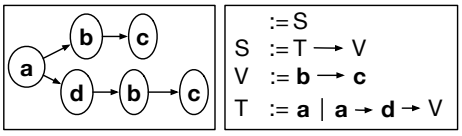


Fig. 4: An example graph on the left and graph grammar production rules on the right which accept that graph. S, T, and V represent non-terminals while a,b,c, and d are terminals.

to a central node would impose unacceptable network cost. This is especially true in the case of clusters that are already deluged with internal network traffic [21].

Winnower's High-level Idea. We observe that applications replicated on multiple nodes will produce highly homogeneous audit logs across executions. As applications will be deployed with nearly-identical configurations (e.g., filesystems, launch sequences), we can expect the resultant provenance graphs to be similar both structurally (i.e., graph connectivity) and semantically (i.e., graph labels). on a per-application (or, per-container) basis. Broadly speaking, our goal is to generate consensus across all nodes to produce a model of application behavior like the one shown in Figure 3b. In contrast to 3a, redundancy between nodes has been eliminated, and each activity is shown only once. However, a *confidence level* marks the level of consensus that was reached between application instances (in 3a MySQL has 5 instances). As the attack occurred on a single node, its confidence level is low, and thus represents anomalous activity that can easily be identified by the administrator. Thus, the consensus model is both efficient to transmit and further, retains the necessary information to identify the attack.

C. Graph Grammars

To facilitate the creation of a cluster-wide provenance model for worker execution, in this work we present a novel adaptation of Discrete Finite Automata (DFA) learning techniques [13]. As DFAs are equivalent to regular grammars [32], this approach is sometimes referred to as *graph grammar* learning. There have been different formulations of graph grammars that broadly refer to classes of standard grammars applied to graphs. In a standard grammar, a language of strings defines a set of rules such that a given string is considered a member of the grammar if it can be constructed from the rules. It is intuitive to extend the notion of standard grammars from strings to graphs, such that a graph belongs to a grammar if it can be constructed from a set of grammatical rules represented in the form of $L := R$ where L is the pattern subgraph (or left-hand side) which can be replaced by R subgraph (or right-hand side). An example of such grammar and a graph that belongs to it is shown in Figure 4.

Graph grammar systems support two important operations: *induction* and *parsing*. Induction algorithms provide a way to learn a new grammar from a set of one or more example graphs. Parsing is a membership test that verifies whether an instance graph can be constructed from a given grammar. Graph grammar learning is not a deterministic process, as multiple grammars can parse the same instance of the graph. As a result, we need heuristic techniques to select an acceptable grammar. While there are strategies for choosing the best grammar during induction, we make use of the Minimum Description Length (MDL) heuristic [40], [17]. The MDL heuristic formalizes the notion that the simplest explanation of data is the best explanation of data. MDL is defined by the following equation:

$$DL(G, S) = DL(S) + DL(G|S) \quad (1)$$

where G is an input graph, S is a model (grammar) of the input graph, $(G|S)$ is G compressed with S, and $DL()$ returns the description length of the input in bits, The MDL heuristic says the best S minimizes $DL(G, S)$; in other words, the optimal S minimizes the space required to represent the input graph.

III. System Design

A. Threat Model & Assumptions

Our approach is designed with consideration for a data center running a distributed application that has been replicated on hundreds or thousands of *Worker* nodes. Workers may run as containers, virtual machines, or bare metal hosts; while our prototype system is implemented for Docker containers (see §IV), our methodology is agnostic to the workers' platform. We require only that each worker is associated with an auditing mechanism that records the actions of the node. In addition to worker nodes, the data center features one *Monitor* node that is operated by a system administrator. The monitor will communicate with worker nodes to obtain audit records of their activities.

The attack surface that we consider in this work is that of the worker nodes. An adversary may attempt to gain remote control of a worker by exploiting a software vulnerability in the distributed application (see §II-A), or may have used a market such as Docker Store to distribute a malicious application that contains a backdoor. Once the attacker gains control of a worker, they may eavesdrop on legitimate user traffic or to make use of the worker's compute resources to perpetrate other misdeeds. In the case of virtualized workers, the attacker's goal may be to break isolation and gain a persistent presence on the underlying machine.

An important consideration for any auditing system is the security of the recording mechanism. This is because it is common practice for system intruders to tamper with audit logs to cover their tracks. While log integrity is an important goal, it is orthogonal to the aims of this system. Therefore, we assume the integrity of the workers' audit mechanisms. In the case of kernel-based audit mechanisms (e.g., `auditd`), kernel hardening techniques (e.g., enabling SELinux) can be deployed to increase the complexity of a successful attack.

B. Design Goals

The limitations outlined in §II-B motivate the following system design goals:

– *Generality.* Winnower design and techniques should be independent of underlying platform (e.g. containers, VM, etc) and applications used by the compute clusters.
– *Minimal Log Transmission.* Winnower should prevent worker nodes from sending redundant audit records to the central node i.e. only transmits the minimum amount

of information required to adequately describe unique or anomalous events within the cluster.
– *Concise Graphs.* Winnower generated provenance graphs on central node should be concise i.e. capturing aggregated cluster-wide activities with any anomalous behaviour visible in graphs.
– *Support Cluster Auditing.* Winnower should support distributed querying worker nodes for complete attack tracing and local policy monitoring in the cluster.

C. System Overview

Winnower acts as a distributed auditing layer that resides on top of individual worker nodes' auditing mechanisms. The core contributions of Winnower are three functions that enable efficient aggregation of audit data at a central monitoring node:

1) *Provenance Graph Abstraction*, in which workers abstract provenance graphs to remove instance-specific and non-deterministic information (§III-D).
2) *Provenance Graph Induction*, in which the worker generates behavioral models and then Monitor aggregates worker models into a single unified model and send them back to all workers (§III-E).
3) *Provenance Model Incremental Updates*, in which workers check to see if newly generated provenance records are described by the global model. If and only if they are not already in the model, the workers transmit the model updates back to the central node (§III-F).

Using aforementioned functions, our aggregation technique works as follows: First, Winnower uses an application-aware provenance tracker on each node to find and separate homogeneous audit logs from replicated applications. In the attack scenario we discussed in §II-B, Winnower separates ProFTPD and MySQL logs. Then, to remove instance-specific information from homogeneous audit logs, Winnower applies provenance graph abstraction function locally on each worker node. In Figure 3a, different IP addresses are present in socket vertices attached to "ftp listener" process vertex. However, exact IP address in vertices is not important to extract behaviour of the application and therefore, we can abstract it before model construction. After that, Winnower applies provenance graph induction function to remove redundancy and generate behavioral models. In Figure 3a, "ftp" process vertex spawns several "ftp listener" process vertices. As they represent semantically equivalent behaviour (causal path is same), we can combine them into a single vertex as shown in Figure 3b. Finally, Winnower prevents worker nodes from transmitting redundant audit records using provenance graph incremental update function and send only the graph grammar model's updates to the central node.

In addition to these core functions, Winnower provides a fully-realized distributed provenance tracing framework that supports forward and backward tracing of system events as well as policy-based automated system monitoring. We describe these features in §IV with greater details.

D. Provenance Graph Abstraction

The core function of Winnower is to ingest the provenance graphs of different worker nodes and output a generic model that summarizes the activity of those nodes. However, even if all nodes are clones of the same image, we can expect that a variety of instance-specific fields and descriptors will be present in each worker's graph. For example, each web service worker will receive web requests from different remote hosts, causing different IP addresses to appear in their provenance graph. We would also expect instance-specific identities assigned to each worker such as a host name or dynamically-assigned IP address. While these details are important when inspecting an individual node, they not useful to an administrator attempting to reason about a distributed system in aggregate. In fact, these instance-specific fields will frustrate any attempts to build a generic application behaviour model because equivalent events have been assigned different labels on different nodes. Therefore, before attempting model generation we must first abstract away these instance-specific details.

To facilitate this abstraction, we group the different fields found in provenance vertex labels into one of three classes, handling each as follows: *equivalence classes* contain instance-specific information and are abstracted using summarization techniques prior to model building; *immutable classes* will not contain instance-specific information and therefore are not changed; finally, *removal classes* are simply removed from the vertex label prior to graph comparison. Below, we explain classification of each field associated with each provenance principle (i.e., activity, artifact, agent).

Activities. Activity vertices consist of five different labels: *Process Name*, *PID*, *Timestamp*, *Current Directory (CWD)*, and *Command line Args*. Because we expect all workers to follow the same general workflow, process name, CWD, and command line arguments are handled as immutable; in other words, a deviation in either of these fields will be visible in the final model. PIDs and Timestamps can both be influenced by non-determinism and vary between executions, and are therefore removed. In Figure 5, `pid` is removed from Activity vertices after graph abstraction step. For brevity, we omit description of other environment variables, which can be handled similarly.

Artifacts. Artifact vertices are further categorized into subtypes based on data types. We describe our general approach with consideration for file and socket artifacts below, omitting other artifacts such as IPC for brevity:

- *File Artifacts*: File subtype vertex consists three labels: *File Path*, *Operation (i.e., read/write/create)* and *Version*. The version field is incremented each time data is written to an argument, which is highly dependent on dynamic events such as network activity and is therefore removed. The operation label is also removed for simplicity, as this information is already encoded in the edge labels of the graph. The most important field, *file path*, is handled differently depending on the class of file: *(a) Core-system Files:* these files are common across all workers and therefore do not need to be abstracted, so we scan the node image to create the set $sysFiles$ and treat these files as immutable. In Figure 5a file path label `/usr/lib/libpq.so` vertex is not removed after abstraction. *(b) Temporary Files:* temporary files are those files who only interact with a single process throughout their entire life cycle. As noted in [51], these files do not have meaning when attack tracing, and can therefore be removed.

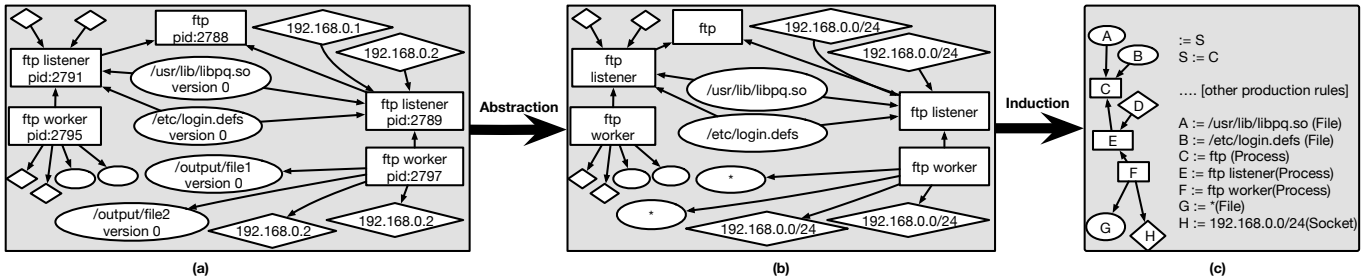


Fig. 5: Applying graph abstraction and graph grammar induction on FTP application provenance graph.

In Figure 5a, artifact file path label `/output/file1` attached to "ftp worker" process vertex is removed after abstraction. *(c) Equivalent Files:* all the other files are treated as the equivalence class. For a given activity, when more than a configurable threshold (τ_{File}) of equivalent files are present, they collapsed into one vertex that is labeled as the most specific common file path across all file paths.

- *Socket Artifacts*: The socket subtype vertex is described by an *IP Address* field. Web services exchange messages over the network with a wide variety of remote clients. The reported IP addresses of the remote clients will lead to many subgraphs within the provenance graph that all describe the same workflow. To provide an easy-to-understand generic model, it is important that the model not grow with the number of remote connections. Therefore, the IP address field is treated as an equivalence class. For a given activity, when more than a configurable threshold (τ_{Sock}) of remote connections are present, they collapsed into one vertex that is labeled as the most specific common subnet across all IP addresses. An example of this is shown in Figure 5a; the artifact that was generated by `ftp worker` represents many network transmissions in the `192.168.0.0/24` subnet mask.

Agents. Agents are described by a *UID* field. Because we expect all workers to follow the same general workflow, we treat UIDs as immutable; in other words, the presence of a new agent on a given node will be visible in the final model.

Graph Abstraction Algorithm. Graph abstraction is triggered by a cluster-wide configurable epoch t, after which each node performs abstraction locally. In Figure 6, we outline the pseudocode for efficient traversal of provenance graph and apply abstraction on each vertex. Because all activities are connected to their child activities, traversing the activity nodes while inspecting their immediate artifact/agent children is sufficient to perform a complete traversal of the provenance DAG. In Figure 6, the functions ABSTRACTACTIVITIES, ABSTRACTFILES, ABSTRACTSOCKETS, and ABSTRACTAGENTS apply the transformations discussed above to the input DAG.

Discussion. Performing the abstractions discussed above will invariably lead to a loss of context in the resulting global model. Eventually, we may need this instance-specific information to perform further attack investigation and incident response. *Therefore, an unmodified record of each worker's provenance (Dag) is maintained on the local node.* An additional concern is that our abstraction techniques can lead to mimicry attacks [66] by launching attack process with the same

Function GRAPHABSTRACT(Dag, $sysFiles$, τ_{File}, τ_{Sock})
  /* Root is always Process Vertex */
  $Root \leftarrow$ Get Root from Dag
  $Dag' \leftarrow Dag$.copy()
  $Queue$.push($Root$)
  while $Queue$ *is not empty* **do**
    $currentVertex \leftarrow Queue$.pop()
    $children \leftarrow currentVertex$.Children()
    $\theta_{files} \leftarrow children$.getFileSubtype()
    $\theta_{socks} \leftarrow children$.getSocketSubtype()
    $\theta_{procs} \leftarrow children$.getProcessSubtype()
    $\theta_{agent} \leftarrow children$.getAgentType()
    $Queue$.push(θ_{procs})
    $Dag' \leftarrow$ ABSTRACTACTIVITIES(Dag',θ_{procs})
    $Dag' \leftarrow$ ABSTRACTFILES(Dag', $sysFiles$,θ_{files}, τ_{File})
    $Dag' \leftarrow$ ABSTRACTSOCKETS(Dag',θ_{socks}, τ_{Sock})
    $Dag' \leftarrow$ ABSTRACTAGENTS(Dag',θ_{agent})
  end
  /* Return Abstracted Provenance DAG */
  return Dag'

Fig. 6: Pseudocode of Provenance Graph Abstraction Function. Functions `getFileSubtype`, `getSocketSubtype`, `getProcessSubtype`, `getAgentType` extract file, socket, process and agent vertices respectively from $children$ list.

name and commandline arguments. However, as we never remove process vertices during abstraction and further, the causal path of attack process vertex will be different, mimicry attacks will always be visible in the final model. Finally, we note that Section V considers the compression benefits of graph abstraction in isolation to our other techniques; abstraction reduces overall log size by roughly 27% in our experiments.

E. Provenance Graph Induction

To generate a global model of worker activity, Winnower makes use of graph grammar learning techniques. However, graph grammars as described in Section II-C are not immediately applicable to provenance graphs. Operations like parsing and induction in prior approaches are prohibitively costly in terms of runtime complexity [41]; this is in part because they consider the general class of directed graphs, in which cycles are common. More importantly, graph grammar techniques are prone to over generalization; in the case of data provenance, this creates the risk of erasing important causal relations between system objects. Consider for example the provenance of the httpd process in Figure 7. Here the `WasTriggeredBy` edges encode an important series of causal relations; However, in experimentation with past techniques we discovered that the learning system would treat all `httpd`

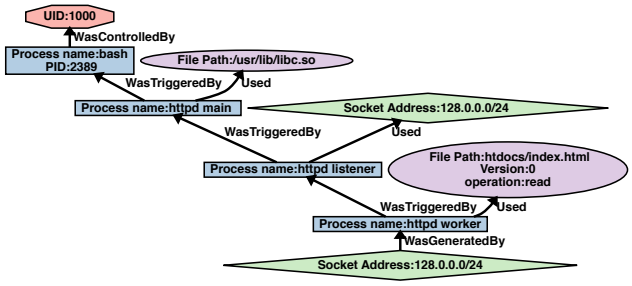


Fig. 7: A simplified provenance graph of an Apache webserver serving a single user request. Past approaches to graph grammar learning would overgeneralize this graph.

`worker` activities as identical, regardless of their ancestry or progeny. In other words, we discovered that the rich contextual information of a provenance graph is difficult to encode as a grammar.

To solve this problem, we adapt techniques from DFA learning. In standard DFA learning [32], [68], the present state of a vertex includes the path taken to reach the vertex. We extend DFA learning to data provenance by defining the state of a vertex not only by its label, but also its prefix state tree (τ_{prefix}) and suffix state tree (τ_{suffix}). A vertex `v`'s prefix state tree is its provenance ancestry – a subgraph describing all the system principles and events that directly or transitively affected by `v`. `v`'s suffix state tree is its provenance progeny – a subgraph describing all of the system principles and events that were in part derived or affected by `v`. In other words, each system object is defined by its label, the system entities that informed its present state, the system entities whose state it informed. In this way, we can be sure that graph induction will retain all causal information needed to describe provenance of system objects. In the example shown in Figure 5b, `ftp worker`'s τ_{prefix} consist of `ftp listener`, `ftp`, `/usr/lib/libpq.so`, and `/etc/login.defs`. Similarly, its τ_{suffix} consists of `192.168.0.0/24` and a summarized * file vertex.

Pseudocode for our graph grammar induction (INDUCTION) function is given in Figure 8, which is MDL-based DFA learning algorithm [13]. We use MDL principle as a guiding metric to choose the best grammar from candidate grammars which minimizes the description cost of the given graph (see §II-C). The algorithm is comprised of the following two steps:

Bootstrapping. In this initial step, a given worker's input provenance graphs $InputDags$ merged by adding a dummy root vertex and joining all the $InputDags$'s root vertices to the dummy root.[4] A single Dag is returned after applying the COMBINEROOTS function. Next, the prefix state tree set τ_{prefix} and suffix state tree set τ_{suffix} are generated for each vertex in Dag, with the results stored in $Gram$. Note that every vertex is uniquely identified by the tuple $\{\tau_{prefix}, \tau_{suffix}, vertexlabel\}$. Further, the set of these combinations for every vertex defines the initial (*specific*) graph grammar $Gram$ for Dag. After bootstrapping, if two vertices are defined by the same tuple they are considered equivalent and merged implicitly in the final grammar. For example, in Figure 5b, the prefix/suffix state trees of the two `ftp worker` vertices are considered the same, and therefore share an entry in $Gram$.

Searching. In this step, the algorithm searches for graph grammars that improve on the naïve initial specific grammar by attempting to minimize the MDL equation 1. The MERGE function applies a "state merging" procedure from DFA learning systems [32], [68]. The main purpose of state merging is to find repetitive structures in the graph and combine them in the grammar. The MERGE function takes two states from the

[4]This step is necessary for our implementation because we make use of a user-space provenance recorder that cannot fully track the system's process tree. A dummy root would not be needed if a whole-system provenance recorder (e.g., [20]) was used instead.

```
Function INDUCTION(InputDags)
    /* Bootstrapping Step                                          */
    Dag ← COMBINEROOTS(InputDags)
    Dag ← TOPOLOGICALSORT(Dag)
    Gram ← ∅
    foreach vertex ∈ Dag do
        τ_prefix ← GETPREFIXTREE(vertex)
        τ_suffix ← GETSUFFIXTREE(vertex)
        Gram ← Gram ∪ {τ_prefix, τ_suffix, vertex.label}
    end
    /* Search Step                                                 */
    Gram_final ← SEARCH(Dag, Gram)
    return Gram_final

Function SEARCH(Dag, Gram)
    cost ← Map from grammar to mdl cost
    cost[Gram] ← MDL(Dag,Gram)
    explore ← PriorityQueue()
    explore.push(Gram)
    while explore is not empty do
        Gram_min ← explore.pop()
        foreach state_1,state_2 ∈ Gram_min do
            Gram_new ← MERGE(Dag, Gram_min,state_1,state_2)
            if Gram_new was not seen then
                cost[Gram_new] ← MDL(Dag,Gram_new)
                explore.push(Gram_new)
            if terminated early then
                /* Final minimum mdl cost grammar                  */
                Gram_final ← GETMIN(cost)
                return Gram_final
        end
    end
    /* Final minimum mdl cost grammar                              */
    Gram_final ← GETMIN(cost)
    return Gram_final

Function PARSE(Dag, Gram)
    Dag ← TOPOLOGICALSORT(Dag)
    G_new ← Gram.copy()
    /* Go through each vertex and confirm the pairing is
       acceptable                                                  */
    foreach vertex ∈ Dag do
        τ_prefix ← GETPREFIXTREE(vertex)
        τ_suffix ← GETSUFFIXTREE(vertex)
        if not ACCEPT(Gram,τ_prefix,τ_suffix,vertex) then
            G_new ← G_new ∪ {τ_prefix, τ_suffix, vertex.label}
    end
    /* Perform search step from induction function on G_new        */
    Gram_final ← SEARCH(Dag, G_new)
    return Gram_final
```

Fig. 8: Pseudocode Graph Grammar Induction and Parsing Functions. Functions `GETPREFIXTREE` and `GETSUFFIXTREE` return prefix and suffix tree of input vertex respectively while Function `GETMIN` returns minimum cost grammar from input map.

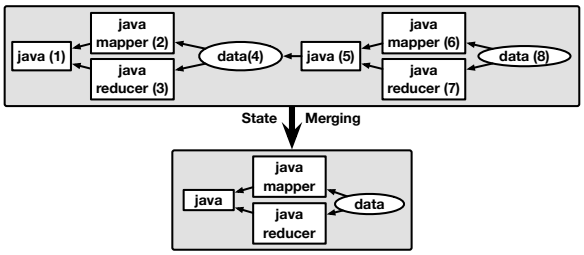


Fig. 9: State merging applied on two chained Hadoop jobs' provenance graph. State merging combines the repetitive subgraphs.

grammar $Gram$ and attempts to make them indistinguishable by merging both τ_{prefix} trees of two selected states, leading to the creation of a new grammar $Gram_{new}$ that remains consistent with the input Dag. Our merge function uses an *evidence driven strategy* [49], which attempts to merge every pair of states from the graph grammar to produce a new candidate graph grammar, To support data provenance, our MERGE function restricts merging of vertices to only those of same type, *e.g.*, process vertices can only be merged with other process vertices.

A thorough description of how state merging works is out of the scope of this paper, we refer readers to [41] for a detailed explanation. For clarity, we provide a simplified example of state merging in Figure 9 that merges the provenance of two chained Hadoop (map/reduce) jobs. The τ_{prefix} of vertex 1 is empty $\{\}$ while τ_{prefix} for vertex 5 consists of vertices $\{1, 2, 3, 4\}$. After vertices 1 and 5 are merged, the τ_{prefix} of 5 becomes empty $\{\}$, and the τ_{prefix} of downstream vertices are also updated to remove $\{1, 2, 3, 4\}$, e.g., vertex 8's τ_{prefix} changes from $\{1, 2, 3, 4, 5\}$ to $\{5\}$. This makes the τ_{prefix} of vertices 5,6,7, and 8 equivalent to the τ_{prefix} of vertices 1,2,3,4, enabling the repetitive subgraphs to be combined into the new grammar. While here we describe state merging for τ_{prefix} only, the process is identical for τ_{suffix}.

The candidate graph grammars from merging step are *general* as they can accept/parse more graphs than the initial (*specific*) graph grammar. Then, the MDL cost of each candidate grammar is calculated using the MDL function according to equation 1, and is added to the $explore$ PriorityQueue. Whenever a new grammar is popped from $explore$, the grammar with the minimal MDL cost is returned. The process of merging states and adding new grammars to $explore$ is repeated until either $explore$ is empty (i.e., convergence is achieved) or the algorithm is terminated by an external decision such as killing the process or exceeding a time limit. In our implementation, we set the termination point when convergence is achieved.

F. Provenance Graph Membership Test and Update

The final component required Winnower is a membership test that, given a grammar and an instance provenance graph, determines whether or not the graph can be produced by the grammar. The membership test algorithm follows naturally from graph grammar parsing algorithms. At a high-level, function PARSE shown in Figure 8 takes as input a graph Dag and a grammar $Gram$. Then, it generates the prefix state tree τ_{prefix} and suffix state tree τ_{prefix} for each vertex in Dag. This step is similar to the bootstrapping step of induction. Finally, the algorithm determines for every vertex of given DAG Dag whether or not its prefix tree state and suffix state tree are present in $Gram$. If input tree state and output state tree of any vertex are not present in $Gram$ then, function ACCEPT returns $False$, meaning that Dag cannot be parsed with $Gram$. Note that parsing in DFA is linear time operation due to its equivalence to regular grammars.

In Winnower, if and only if parsing fails, it is necessary for the worker to transmit additional provenance records to the Monitor. To do so, the worker updates $Gram$ to incorporate the instance Dag by adding the unparsable vertices to it. It then generates a new grammar by locally invoking the SEARCH step of the INDUCTION function. The resulting new grammar $Gram_{final}$, is then transmitted to the Monitor.

IV. SYSTEM IMPLEMENTATION

We have implemented a prototype version of Winnower for Linux OS with Docker Swarm version 1.2.6 and Docker version 1.13. An architectural overview of the Winnower system is shown in Figure 10. A complete workflow for how Winnower enables auditing and attack investigation is as follows: ① a provenance graph for each container is generated by the host machine using `auditd`; ② a Winnower client agent running on each worker node applies graph grammar induction locally to produce a local model that is pushed to the central Monitor; ③ the central Winnower monitor performs induction to unify the local models from all worker nodes into a single global model, maintaining a confidence ζ value for each vertex representing how many workers reported each behavior, then transmits the global model back to the worker nodes; ④ administrators can quickly view anomalous activities (i.e., vertices with low ζ values) and decide whether to investigate; ⑤ during an investigation, the administrator can issue queries to the Winnower monitor, or ⑥ request a complete (unabstracted) copy of the workers high-fidelity provenance graph, which is maintained on the worker nodes. This final step is necessary to ensure that no important forensic context is lost during the model generation.

Worker Components. Winnower requires `auditd` to be enabled on all connected worker nodes, as well as SELinux[5]. `SVirt` [57] runs each Docker container in its own SELinux contexts if the docker daemon is started with option *–selinux-enabled*. The Docker daemon generates unique SELinux labels called Multi-Category Security (MCS) labels and assigns them to each process, file, and network socket of a specific container. Finally, Winnower workers run a modified version of the SPADE system [37], which parses `auditd` logs and generates causal dependencies in the form of OPM-compliant provenance graphs [39]. While our prototype makes use of SPADE for ease-of-deployment, the provenance recorder used by Winnower is largely modular and could be quickly replaced by a kernel-level provenance recorder [20], [54], [61] to achieve stronger security or completeness guarantees

When a system call occurs on the worker, the execution and associated call arguments are captured by `auditd` based on

[5]SELinux is a Linux kernel feature that allows fine-grained restrictions on application permissions. In an SELinux enabled OS, each process has an associated context, and a set of rules define the interactions permitted between contexts. This allows strict limits to be placed on how processes can interact and which resources they can access.

the rules defined in `/etc/audit/audit.rules`. Winnower uses all the `audit.rules` to capture the syscalls events that can be useful in attack tracing, such as process, file, and socket manipulation. After the syscall is processed by the kernel, `auditd` sends data from the kernel to the user-space daemon which writes the event to a persistent log file. `auditd` writes SELinux labels along with other event information such as process id into the logs. To differentiate the provenance of different containers, Winnower extends SPADE to communicate with the Docker Swarm and map each objects' SELinux labels to the associated `container-id` and `image-id` given by Docker to find containers belonging to same applications. Winnower then uses SPADE's Graphviz[6] backend to record container-specific provenance graphs and performs DFA learning over the resulting `dot` files.

The Winnower agent runs locally on each worker node in the cluster and communicates with the Monitor's Winnower frontend. After performing graph abstraction and local induction as discussed in Sections III-D and III-E, it is responsible for publishing local models to the Winnower frontend via a publisher-subscriber model at a configurable interval (epoch). We used Apache Kafka pub/sub system. The Winnower agent also waits for instructions from the Winnower frontend related to provenance queries, changes in epoch size, or deploying provenance policies. After each epoch t, the Winnower performs graph grammar induction on the worker's current provenance graph.

Monitor Components. The Monitor node is responsible for running the Docker Swarm manager, and is extended by Winnower to run a frontend consisting of five submodules: *1)* a *Provenance Manager* submodule gathers provenance graphs from each worker node and sends back the current globally aggregated model. *2)* a *Provenance Query* submodule that supports forward and backward tracing queries. The three functions provided by Winnower to support tracing are shown in the Table I. The user first identifies nodes of interest with the `getNodes` by specifying a key-value pair (e.g., key="name", value="index.html"). These node IDs can then be passed to the `getAncestors` or `getDescendants` functions to perform backward and forward tracing, respectively. To track the migration of workers in dynamic scheduling environments, Winnower maintains a log of the scheduling decisions made by Docker Swarm and transparently identifies which nodes to query to reconstitute the full provenance graph. *3)* a *Policy Engine* submodule exposes a simple Cypher-like [12] graph query language that permits administrators to define automated responses when a specified property is detected within a worker's provenance graph. The format of policy is shown in the Figure 11. In the `MATCH` clause the pattern to match is given while `RETURN` will send matched vertex id to administrator to run forward/backward queries. Figure 12 shows an example policy. Here, if any process writes to the `/usr/bin/` directory on a worker node, the administrator will be notified. *4)* finally, a *Docker API Calls* submodule uses Docker Swarm API to get information regarding containers in cluster such as which containers belong to same information, and liveness of containers. *5)* a final component of the Winnower frontend is the *Provenance Learning* submodule. After each epoch t, the Provenance Manager fetches new provenance graph from workers and parses them into the scala graph format using the scalax package [3]. The Provenance Learning submodule first checks if it already has the graph grammar model for the worker. If there is a model previously generated then it will be updated through induction to incorporate the new graphs Otherwise the provenance learning system merges the worker graph model from the current epoch into a single global model, then sends them back to each worker. We have implemented provenance graph grammar learning framework in Scala with 3K LOC.

[6]Available at http://www.graphviz.org/

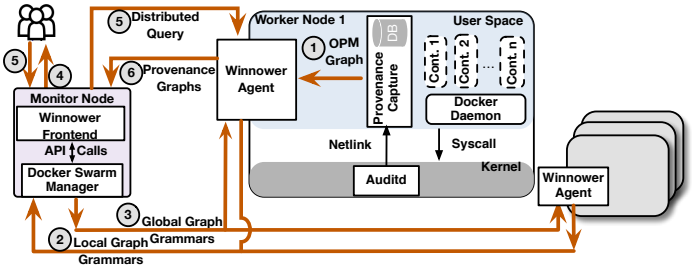


Fig. 10: Winnower Architecture and Workflow (§IV).

TABLE I: Winnower API functions for attack tracing on provenance graphs generated from graph grammars models.

`getNodes(key, value)` → `node_ids`
`getAncestors(node_id)` → `graph`
`getDescendants(node_id)` → `graph`

```
MATCH vertex_a:{labels} edge_a
      vertex_b:{ labels } edge_b
      ...
RETURN vertex_a.id
```

Fig. 11: Format of Provenance Policy Language to check certain provenance DAG pattern on each worker node.

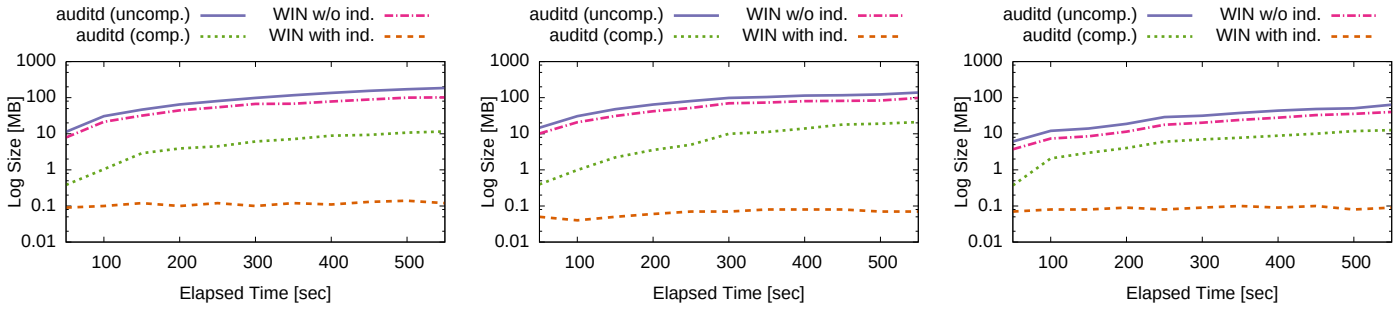
```
MATCH (a:Process {name:"*"}) used
      (b: File { file path:"/usr/bin/*", operation:"write"})
RETURN a.id
```

Fig. 12: Example of Provenance Policy which monitors any process writing to /usr/bin directory.

V. Performance

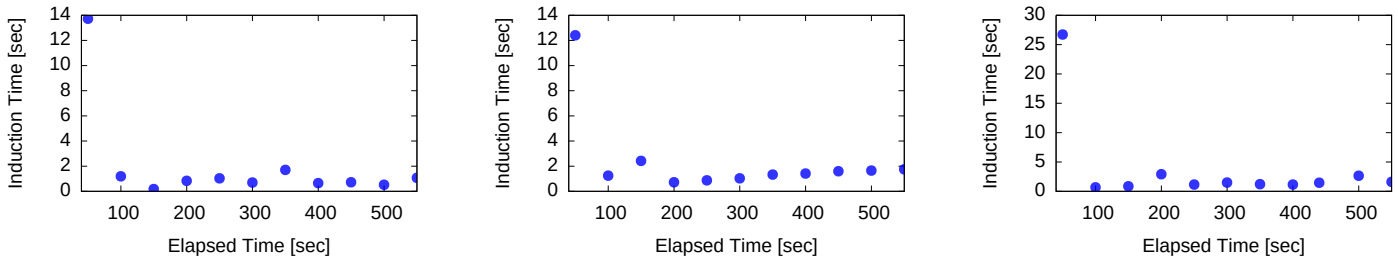
In order to evaluate the performance of Winnower, we profiled 3 popular server applications on a five node cluster using Docker Swarm. Workloads were generated for these applications using the standard benchmarking tools Apache Benchmark `ab`[7], FTPbench[8], and SysBench[9] which were also used in most relevant prior work [51], [50], [54]. The cluster was deployed as KVM/QEMU virtual machines on a single server running Ubuntu 16.04 LTS with 20 Intel Xeon (E5-2630) CPUs with 64 GB RAM. One VM in the cluster acted as the Monitor, running the Winnower Frontend and Docker Swarm manager, while the remaining four VMs hosted worker containers. Each VM had 2 VCPUs, 4GB RAM, and ran CentOS 7. We deployed total 20 application containers for

[7]Available at https://httpd.apache.org/docs/2.4/programs/ab.html
[8]Available at https://pypi.python.org/pypi/ftpbench
[9]Available at https://dev.mysql.com/downloads/benchmarks.html



(a) ProFTPD **(b)** Apache **(c)** MySQL

Fig. 13: Average accumulated audit log size (log scaled) on central node overtime for each application. Winnower generated audit logs (WIN) with and without induction step for each application are substantially less than `auditd`/SPADE log (compressed and uncompressed).



(a) ProFTPD **(b)** Apache **(c)** MySQL

Fig. 14: Average time spent on graph grammar induction and parsing at each epoch, which occurred every 50 seconds.

each benchmark across the cluster. For each workload, the Monitor sends requests with the concurrency-level parameter of 40 that were load balanced across the worker nodes. Note that the total number of requests depends on how long the benchmark was run. Winnower was configured with an epoch size of 50 seconds, with set τ_{File} and τ_{Sock} thresholds to 400. To serve as a baseline comparison for Winnower, we set-up daemons on each worker that stream `auditd` activity to the Monitor node.

Our performance evaluation sets out to answer the following questions about Winnower:

– §V-A: What is the overall storage reduction provided by Winnower' abstraction and induction techniques? To answer this question, for each workload we compare the compressed and uncompressed size of `auditd` logs to the size of our Winnower model under two configurations: abstraction only (*WIN w/o ind.*) and abstraction/induction (*WIN with ind.*).
– §V-B: What is the computational cost of generating a Winnower model? To answer this question, we measure the induction speed for each epoch in each workload.
– §V-C: What is the network cost of operating Winnower? To answer this question, we compare the cost of transmitting Winnower models over the network to the configuration in which all `auditd` activity is streamed to the Monitor.

A. Storage Reduction

For each workload, we measured the storage requirements at the Monitor node for both Winnower and `auditd`. Figure 13 shows the space overhead over time for Winnower as compared to `auditd`; note that the y-axis uses a log scale. Table II provides a total summary of space overhead and graph complexity

		# of Vertices		# of Edges		Log Size (MB)	
App	**Duration**	ASD	WIN	ASD	WIN	ASD	WIN
Apache	33m12s	1.04m	32	1.04m	41	485	0.11
ProFTPD	20m12s	340k	56	340k	58	630	0.12
MySQL	21m00s	840k	61	840k	64	130	0.17

TABLE II: Summary of observed space overheads in test applications comparing `auditd`/SPADE (ASD) to Winnower (WIN). Winnower consistently reduces storage costs by over three orders of magnitude.

for all three applications. The graph abstraction step (*WIN w/o ind.*) accounts for only a small amount of compression compared to graph induction, but enables the effectiveness of induction as discussed in §III-D. Approximately 0.6 GB of data per hour is generated by `auditd`/SPADE (*auditd(uncomp.)*) on central node, in contrast to 150KB per hour by Winnower. With graph induction enabled (*WIN with ind.*), *Winnower outperforms `auditd` by 3 orders of magnitude, reducing the storage burden by 99.9%.* Even when `auditd` output is compressed with `7z` tool at the Monitor, Winnower still reduces the storage burden by 99.2%. Winnower thus dramatically reduces storage cost at the administrative node.

B. Computational Cost

For each workload, Figure 14 shows the time spent on graph induction on each node after each 50 second epoch; that is, each node ingested the 50 seconds of log data, then performed the graph grammar inference algorithm (§III-E) to generate a provenance model. We observe that induction during the first epoch is more costly (12-26 seconds) than subsequent epochs (0-3 seconds). This is because a significantly larger amount of graph structure is learned during the initial

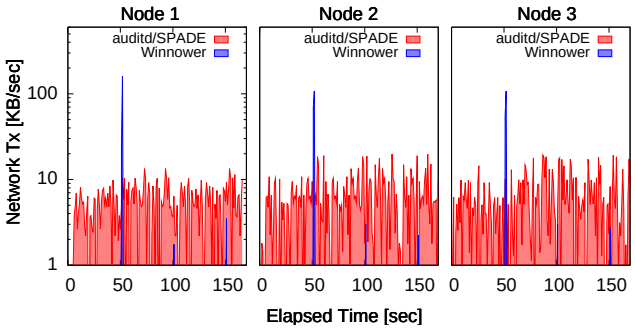


Fig. 15: Network throughput (log scaled) for transmitting provenance logs to central node over time on 3 nodes during our experiments.

grammatical inference, whereas in subsequent inductions the structure of the model is quite stable and only incremental updates occur. As we noted in §III-F that parsing is a linear time operation we omit parsing computation cost for brevity.

Setting aside the initial induction, these results show that the smallest safe epoch size for our current implementation is about 5 seconds. This value represents an upper bound on the frequency with which the provenance model can be updated. However, we are confident that smaller epochs could be supported through optimizing our prototype. Specifically, we are currently investigating on re-implementing our induction algorithm as a parallelizable C/C++ program.

C. Network Activity

Finally, we profiled the network activity of Winnower as compared to `auditd` for each workflow. Our results are shown in Figure 15 for MySQL benchmark. Other application benchmarks follow the same trend. Following the first epoch, Winnower transmits a model that summarizes the activities of the first 50 seconds, leading to a brief spike in network transmission. It is important to acknowledge that this behavior could lead to minor performance issues during the initial deployment of Winnower. However, following the first epoch Winnower visibly outperforms `auditd`/SPADE. Over the course of the entire test (21 mins), Winnower transmits just 178KB of data compared to 130MB by `auditd`/SPADE in the whole cluster. Winnower thus offers a dramatic improvement over `auditd`, which continually transmits redundant audit data and may even saturate network links in larger clusters.

VI. Case Studies

In this section, we will demonstrate the efficacy of Winnower in assisting attack investigation by considering five real-world attacks against a Docker Swarm cluster. For each scenario, we setup the five node cluster as used in §V, with one node acting as the monitor and the other four acting as worker nodes. We then ran a series of different multi-container applications for a period of time before launching an attack. We first generated the concise provenance model using Winnower, then determined if it was adequate for attack investigation by performing forward and backward tracing over the graph. To ensure the *completeness* of Winnower, we also repeated each trial using `auditd`/SPADE and compared the two results.

A summary of our findings is shown in Table III. In addition to recording adequate context to explain all attack

		Log Size (MB)		Query Resp. Time (ms)	
Scenario	**Duration**	ASD	WIN	ASD	WIN
ImageTragick Attack	10m12s	231	0.3	103	5
Ransomware Attack	7m21s	161	0.7	102	9
Inexperienced Admin	2m40s	228	0.8	68	4
Dirty Cow Attack	4m21s	301	19	107	12
Backdoor Attack	19m31s	133	0.2	135	5

TABLE III: Summary of Winnower performance in attack scenarios. Winnower (WIN) again reduces log size by three orders of magnitude compared to `auditd`/SPADE (ASD), and improves query performance by two orders of magnitude.

scenarios, Winnower is also able to respond to queries in just a handful of milliseconds, compared to hundreds of milliseconds for `auditd`. Figures 16-18 visualize the models demonstrated by Winnower in each scenario. For clarity, we have annotated each model to draw the reader's attention to the attack; however, please note that the boxed subgraph corresponds perfectly to the confidence level legend, such that an administrator would be able to accurately interpret the graph even without this annotation.

A. ImageTragick Attack

Scenario. We first consider an image manipulation webservice that allows users to perform different operations on uploaded images such as crop and resize. We created this webservice with 10 Nginx webserver docker containers for sending/receiving web requests and 10 ImageMagick containers for image manipulation. Unfortunately, the image manipulation workers were susceptible to the ImageTragick[10] attack. After sending heterogeneous requests for some period, we initiated the attack by uploading a malicious image file `mal.jpg` capable of opening a remote shell back to the attacker's host. The uploaded file contained the following payload:

```
image over 0,0 0,0 'https://127.0.0.1/x.php?x=`bash -i >\&
/dev/tcp/X.X.X.X/9999 0>\&1`'
```

The server executes this code when the image is processed by the `identify` tool of the ImageMagick library, causing a bash shell to be linked to the attacker's remote host.

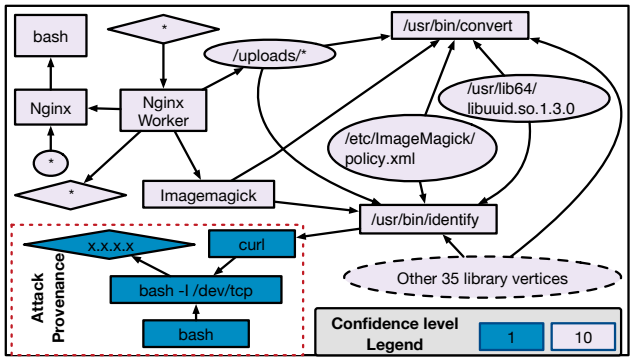


Fig. 16: The concise provenance graph generated by Winnower for imagetragick attack.

Detection. Figure 16 shows the monitor node's view of the

[10] Available at https://imagetragick.com/

attack as provided by the Winnower provenance model. The provenance graph generated by Winnower is remarkably concise, allowing the administrator to easily spot the anomalous activities annotated by the dashed line (Attack Provenance). On the other hand with the auditd/SPADE, the administrator would have had to navigate a provenance graph of 64,811 vertices in order to detect and investigate the attack.

B. Ransomware Attack

Scenario. In this scenario, we consider a Ransomware attack[11] against a vulnerable version (<3.2.0) of the Redis database. The attack exploits a vulnerability that permits an attacker to view and modify the database by executing a `CONFIG` command on an open TCP port. We created an online storage service using Nginx webserver backed by Redis-3.0.0 with sharded database. All Redis containers had public IP assigned, but one of service was permitted to run on a default port, which allowed an attacker to find the vulnerable instance through internet-wide scanning. We generated a workload for the webservice by uploading and downloading content from the site, then executed a ransomware attack: the attacker first connects directly to Redis container over the default port, executes the `Flushall` command to erase the whole database, uploads their SSH key to the database, then obtains root access to the container by using `CONFIG` to copy the database to the root's `.ssh` directory and renames it to `authorized_keys`. After obtaining root access, the attacker connects and leaves a note in the home directory asking for bitcoins to get the encrypted database back.

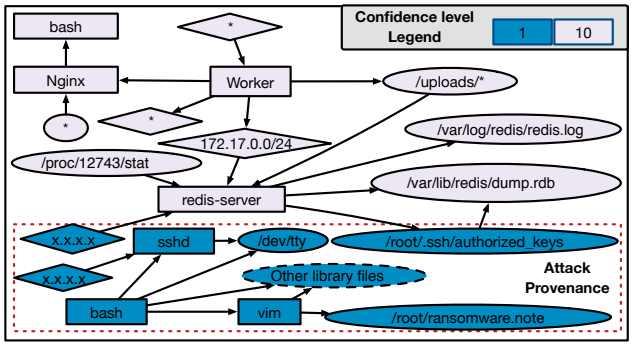


Fig. 17: The provenance graph generated by Winnower for ransomware attack.

Detection. Winnower's utility in this scenario is two-fold. First, as Winnower generates a concise provenance model as shown in Figure 17, the administrator will be able to quickly identify the malicious activity on the cluster, potentially preventing the attack from spreading to the other containers. In contrast, the raw provenance graph generated by auditd/SPADE have 78,149 vertices. Second, by using the complete attack provenance, the administrator will be able to see that the database was not actually sent to internet or encrypted, meaning that this was a fake ransomware attack and the data was irrevocably lost.

[11] See https://duo.com/blog/over-18000-redis-instances-targeted-by-fake-ransomware

C. Inexperienced Administrator

Scenario. In this case study, we consider the inexperienced administrator of a Hadoop container cluster that runs analysis jobs on different datasets[12]. The admin of the cluster left the Docker daemon REST API TCP port open to the Internet, permitting any remote user to create and run a container in the cluster[13]. An attacker can run a reverse TCP shell from the container, then use the container for malicious purposes such as DDoS attacks. In this scenario, we spawned 10 Hadoop containers executing a built-in example of distributed grep on different datasets, then launched a reverse shell attack.

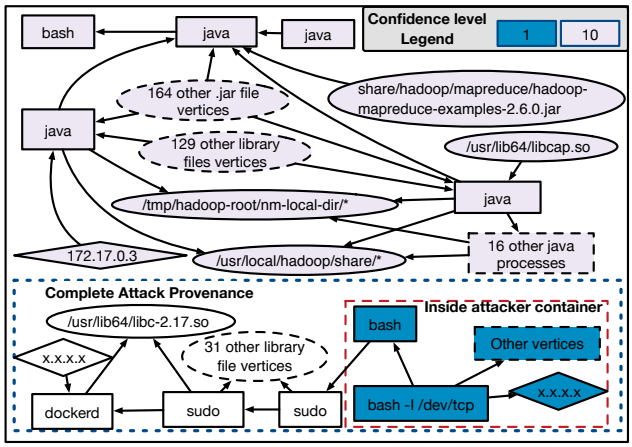


Fig. 18: The provenance graph generated by Winnower for inexperienced administrator case study. We have simplified this diagram for readability.

Investigation. The provenance model generated by Winnower is shown in Figure 18. We have simplified this diagram for readability by making dashed line vertices for different library/System files; regardless, the graph is concise with 319 vertices as compared to auditd/SPADE, which generated a graph of 87,345 vertices. The administrator can easily see in the Winnower model that one container is acting differently than the other workers in the cluster. The admin can then run a backward tracing query on the suspicious vertex to produce a complete explanation of the attack, trail as annotated by blue dashed line, to identify the open Docker port as the method of entry.

D. Dirty Cow Attack

Scenario. In this case study, we consider an online Distributed Continuous Integration (CI) service such as Travis CI[14] or AppVeyor[15] which automatically build and test projects. Users can provide custom scripts which install all dependencies (e.g. `maven`, etc.) and build the project. These services provide users with a terminal-like interface to view all the logging output generated during the project's build process. Consider a CI service that uses Docker and spans a new container for

[12] Available at https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/DockerContainerExecutor.html

[13] See https://threatpost.com/attack-uses-docker-containers-to-hide-persist-plant-malware/126992/

[14] Available at https://travis-ci.org/

[15] Available at https://www.appveyor.com/

each build, which is the case for Travis CI. Here, the CI administrator needs to make sure that the user is never able to break the isolation of the container, as this would allow them to view source code from other (possibly proprietary) builds. However, the base image (e.g., Ubuntu, CentOS) used by this CI service is vulnerable to the Dirty Cow privilege escalation bug (CVE-2016-5195) [16]. Since any user can upload custom bash scripts on the CI service this bug can be exploited to escape isolation and read other users source code [6]. To setup the CI service, we created 10 Docker containers with vulnerable base image kernels. We then ran the build processes of different open source maven-based java projects from Github. During one of the builds, we executed a Dirty Cow attack script from [8].

```
MATCH (a:Agent {UID="0"}) WasControlledBy
      (a:Process {name:"/bin/sh"})
```

Fig. 19: Provenance Policy which will be deployed on each worker node in the cluster to monitor container breakout attacks.

Monitoring. As it is possible to express a breach of container isolation in policy language, this scenario shows Winnower's utility as an active monitoring tool in addition to an administrative aid. As there is no condition under which a container should interact with system objects outside of the container, the administrator can define a provenance policy on each worker node like the one shown in Figure 19. This policy is matched when there is some `/bin/sh` process controlled by UID 0. If the policy is triggered at runtime, a notification is sent to the administrator. Once the administrator has been notified they can run backward tracing query on the bash vertex to reconstruct the attack, which will aid in identifying the vulnerable base image. One might argue that the CI service could block all ports by using SELinux to stop such behaviour; however, CI services cannot do that because they provide software testing services that may require access to these ports. In this attack scenario, since each container in the cluster was building a different project, Winnower does not provide a significant decrease in log size, as shown in Table III. In order for Winnower to work as a compression mechanism, it would be necessary to maintain a separate provenance model for each project, which would eliminate audit redundancy over sequential builds.

E. Backdoor attack.

We describe this attack in §II-B and visualize the concise provenance model in Figure 3b. Our results compared to the `auditd`/SPADE are shown in Table III.

VII. Related Work

In Section II-B we described the limitations with existing provenance collection tools that Winnower addresses, and complement the discussion on related work here.

System-level Provenance. To the best of our knowledge, this is the first work to study an efficient system-level provenance collection mechanism for clusters and solve the challenges

[16]Dirty Cow Bug is a privilege escalation bug in the Linux Kernel discovered on October 20th, 2016. It stems from a race condition in the way that the Linux kernel's memory subsystem handles read only private mappings when a Copy On Write situation is triggered.

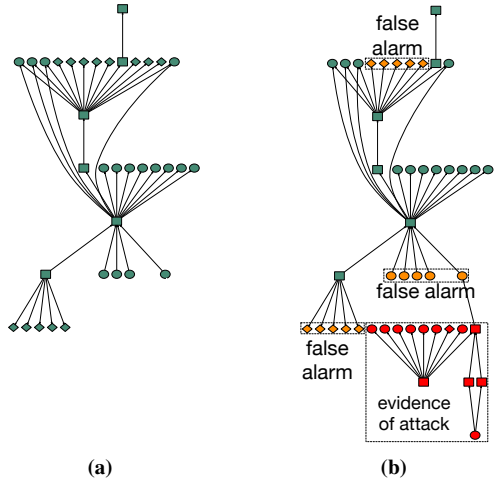


Fig. 20: Provenance of two `httpd` server executions. **(a)** shows normal execution, but execution **(b)** shows evidence of an attack. Comparing these provenance graphs with existing techniques leads to *false alarms*, a limitation that we address with Winnower.

associated with it. However, in recent years, a significant progress has been made to capture system-level provenance and leverage them for forensics [47], [61], [46], [20], [60], [37], [67]. Winnower complements all these OS-level logging systems. However, using existing system logs accumulate very quickly; which makes them impractical to collect and query logs for the scale of clusters. Winnower applies novel graph grammars approach which substantially reduces the cost of storing and processing logs. LogGC [51] provides offline techniques to garbage collect redundant events which have no forensic value. These techniques can be applied alongside our model construction to further decrease storage overheads. Finally, our work also complements the execution partitioning systems such as BEEP/MPI [50], [53] which improve post-mortem analysis by solving the problem of dependency explosion. Liu *et al*. [52] proposed PrioTracker which accelerates the attack causality analysis by adding priority to rare events in attack graph construction. Winnower can be used along with PrioTracker to reduce the overhead of storing and transmitting provenance graphs in a large distributed system before attack causality analysis.

Distributed System Tracing. Existing academic tools [55], [64], [33], [16] and commercial tools [11], [10] on distributed system tracing are mainly targeted towards runtime profiling, finding limping hardware and software misconfigurations. Pinpoint [29] collects execution traces as paths through the system and diagnose anomalies by generating a probabilistic context-free grammar from the paths. However, these systems do not provide *causal relationships* between kernel-level events which is necessary for security auditing and forensics. Moreover, they also suffer from the challenges of log storage/transmission overhead on central node which are outlined in §II-B.

Graph Comparison Techniques. Winnower leverages graph comparison techniques to identify similarities and differences between provenance graphs across different executions. Graph comparison algorithms accept two graphs G and G' as input and output a value quantifying the similarity/difference between these two input graphs [62], [42], [24]. For our

purposes, existing graph comparisons solutions are not immediately applicable, in part because of various sources of non-determinism lead to subtle structural and semantic variations between provenance graphs across executions. To illustrate this limitation, we note briefly a preliminary experiment that we conducted using naïve graph diff in place of DFA learning. Figure 20 shows simplified provenance graphs for two Apache `httpd` webservers, one of which (20b) has fallen victim to a a reverse shell invocation attack. The naive graph diff flagged several subgraphs as anomalous, although they described the same behavior in both executions. In contrast, Winnower accurately identifies the attack subgraph without false alarms.

Previous studies have used graph grammar techniques to infer program behavior and specifications using syscall and function call graphs [43], [31]. Babic *et al*. [14] used induction on tree grammar to learn malicious syscall patterns which they hoped to recognize at runtime. One of the prominent works in the graph grammar learning space is Jonyer *et al*. [44] SubDue system that generates context free grammars to help solve Frequent Subgraph Mining. In light of the high overheads associated with DFA learning, Winnower considers a significantly more challenging problem of how to leverage these techniques in a real time distributed monitoring architecture. Moreover, we also demonstrate methods of abstracting instance-specific details out of audit records to further improve the compression rate of graph induction.

Deduplication and Compression. Our work is orthogonal to provenance graph compression and deduplication techniques [69], [25], [15] due to distributed setting of system-level provenance in our work. Winnower provides scalable compression using DFA learning that exploits the homogeneity present in same applications' provenance across different executions to remove redundancy and generate DFA models. Moreover, DFA models provide an efficient means of membership test which is leveraged by Winnower to avoid redundant transmission of provenance data to the central node. In contrast, deduplication and compression techniques do not provide these functionalities. Recently, Chen *et al*. [28] proposed equivalence-based provenance tree compression to reduce storage overhead. However, their proposal requires distributed applications to be written in a new domain-specific language to find equivalent trees at compile time and works only for network provenance trees.

VIII. Discussion

Our techniques of graph abstraction and DFA learning for system-level provenance are generic; they can be employed in other domains in which there is redundancy across executions, such as multiple VMs or independent process executions as we do not make any assumptions regarding application and their workloads. We focus on container clusters in this paper because these techniques are ideal for environments that adhere to the microservice architecture principle (software as discrete, loosely-coupled, replicated services for scalability and fault-tolerance) and there is a recent paradigm shift in industry towards using Docker containers in the clusters due it's advantages over hypervisor-based VMs [34].

Our framework is extensible to Kubernetes [2], another popular container cluster management tool. The only sub-module from our architecture (§IV) that needs to be changed is the Docker API call stack (consists of 150 LoC) which polls Docker Swarm for different operations such as checking container liveness and finding containers belong to same application. Moreover, Kubernetes also allows creating containers with SELinux labels by defining `seLinuxOptions` field in container manifest file.

IX. Conclusion

In this work, we present Winnower, the first practical system for end-to-end provenance-based auditing of clusters at scale. Winnower includes a novel adaptation of graph abstraction techniques that removes instance-specific information from the system-level provenance graphs and further apply graph grammar principles that enables efficient behavioural modeling and comparison in the provenance domain. We evaluated Winnower performance on cluster applications and five real-world attack scenarios and show that Winnower reduces storage and network overheads by several orders of magnitude compared with existing solutions, while preserving the necessary context to identify and investigate system attacks.

Acknowledgments

We would like to thank Hassaan Irshad (SRI International) for his assistance with the SPADE tool and Ben Rabe for his help in running some experiments. We also thank the anonymous reviewers for their helpful feedback. This work was supported in part by NSF CNS grant 16-57534. The views expressed are those of the authors only.

References

[1] "Docker Company," https://www.docker.com/company.

[2] "Production-grade container orchestration," https://kubernetes.io/.

[3] "Graph For Scala," http://www.scala-graph.org/.

[4] "Docker Store - Find Trusted and Enterprise Ready Containers," https://www.store.docker.com/.

[5] "System administration utilities," man7.org/linux/man-pages/man8/auditd.8.html/.

[6] "Dirty COW Vulnerability: Impact on Containers," http://blog.aquasec.com/dirty-cow-vulnerability-impact-on-containers.

[7] "Anchore Example Queries," https://github.com/anchore/anchore/wiki/Example-Queries.

[8] "Dirtycow-docker-vdso," https://github.com/gebl/dirtycow-docker-vdso.

[9] "The State Of Containerization: Survey," http://red.ht/2iortcK.

[10] "LTTng," http://lttng.org.

[11] "Sysdig," https://www.sysdig.org.

[12] "openCypher," http://www.opencypher.org/.

[13] P. Adriaans and C. Jacobs, "Using MDL for grammar induction," *Grammatical Inference: Algorithms and* ..., pp. 293–306, 2006.

[14] D. Babić, D. Reynaud, and D. Song, "Recognizing malicious software behaviors with tree automata inference," *Form. Methods Syst. Des.*, vol. 41, no. 1, 2012.

[15] Z. Bao, H. Köhler, L. Wang, X. Zhou, and S. Sadiq, "Efficient provenance storage for relational queries," in *CIKM*, 2012.

[16] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using magpie for request extraction and workload modelling." in *OSDI*, 2004.

[17] A. Barron, J. Rissanen, and B. Yu, "The minimum description length principle in coding and modeling," *IEEE Transactions on Information Theory*, vol. 44, no. 6, pp. 2743–2760, 1998.

[18] A. Bates, K. Butler, A. Haeberlen, M. Sherr, and W. Zhou, "Let SDN Be Your Eyes: Secure Forensics in Data Center Networks," in *SENT*, 2014.

[19] A. Bates, W. U. Hassan, K. Butler, A. Dobra, B. Reaves, P. Cable, T. Moyer, and N. Schear, "Transparent web service auditing via network provenance functions," in *WWW*, 2017.

[20] A. Bates, D. Tian, K. R. B. Butler, and T. Moyer, "Trustworthy whole-system provenance for the linux kernel," in *USENIX Security*, 2015.

[21] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *IMC*, 2010.

[22] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[23] M. A. Borkin, C. S. Yeh, M. Boyd, P. Macko, K. Z. Gajos, M. Seltzer, and H. Pfister, "Evaluation of filesystem provenance visualization tools," *IEEE Transactions on Visualization and Computer Graphics*, vol. 19, no. 12, pp. 2476–2485, 2013.

[24] H. Bunke, "On a relation between graph edit distance and maximum common subgraph," *Pattern Recognition Letters*, vol. 18, no. 8, 1997.

[25] A. Chapman, H. Jagadish, and P. Ramanan, "Efficient Provenance Storage," in *SIGMOD*, 2008.

[26] A. Chen, Y. Wu, A. Haeberlen, B. T. Loo, and W. Zhou, "Data provenance at internet scale: Architecture, experiences, and the road ahead," in *CIDR*, 2017.

[27] A. Chen, Y. Wu, A. Haeberlen, W. Zhou, and B. T. Loo, "The good, the bad, and the differences: Better network diagnostics with differential provenance," in *SIGCOMM*, 2016.

[28] C. Chen, H. T. Lehri, L. Kuan Loh, A. Alur, L. Jia, B. T. Loo, and W. Zhou, "Distributed provenance compression," in *SIGMOD*, 2017.

[29] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer, "Path-based faliure and evolution management," in *NSDI*, 2004.

[30] L. Chiticariu, W.-C. Tan, and G. Vijayvargiya, "DBNotes: A Post-it System for Relational Databases Based on Provenance," in *SIGMOD*, 2005.

[31] J. E. Cook and A. L. Wolf, "Discovering models of software processes from event-based data," *ACM Trans. Softw. Eng. Methodol.*, vol. 7, no. 3, pp. 215–249, 1998.

[32] C. De la Higuera, *Grammatical inference: learning automata and grammars*. Cambridge University Press, 2010.

[33] U. Erlingsson, M. Peinado, S. Peter, M. Budiu, and G. Mainar-Ruiz, "Fay: Extensible distributed tracing from kernels to clusters," *ACM Trans. Comput. Syst.*, vol. 30, pp. 13:1–13:35, 2012.

[34] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *ISPASS*, 2015.

[35] I. T. Foster, J.-S. Vöckler, M. Wilde, and Y. Zhao, "Chimera: AVirtual Data System for Representing, Querying, and Automating Data Derivation," in *SSDBM*, 2002.

[36] A. Gehani, H. Kazmi, and H. Irshad, "Scaling SPADE to "Big Provenance"," in *TaPP*, 2016.

[37] A. Gehani and D. Tariq, "SPADE: Support for Provenance Auditing in Distributed Environments," in *Middleware*, 2012.

[38] A. Goel, K. Po, K. Farhadi, Z. Li, and E. De Lara, "The taser intrusion recovery system," in *ACM SIGOPS Operating Systems Review*, 2005.

[39] P. Groth and L. Moreau, "Representing Distributed Systems Using the Open Provenance Model," *Future Gener. Comput. Syst.*, vol. 27, no. 6, 2011.

[40] P. D. Grünwald, *The minimum description length principle*. MIT press, 2007.

[41] C. D. L. Higuera, "A bibliographical study of grammatical inference," *Pattern recognition*, 2005.

[42] C. Jiang, F. Coenen, and M. Zito, "A survey of frequent subgraph mining algorithms," *The Knowledge Engineering Review*, vol. 28, no. 01, pp. 75–105, 2013.

[43] R. Jin, C. Wang, D. Polshakov, S. Parthasarathy, and G. Agrawal, "Discovering frequent topological structures from graph datasets," in *KDD*, 2005.

[44] I. Jonyer, L. B. Holder, and D. J. Cook, "MDL-based context-free graph grammar induction and applications," *International Journal on Artificial Intelligence Tools*, vol. 13, no. 01, pp. 65–79, 2004.

[45] G. Karvounarakis, T. J. Green, Z. G. Ives, and V. Tannen, "Collaborative data sharing via update exchange and provenance," *ACM Trans. Database Syst.*, vol. 38, no. 3, pp. 19:1–19:42, 2013.

[46] T. Kim, X. Wang, N. Zeldovich, and M. F. Kaashoek, "Intrusion recovery using selective re-execution," in *OSDI*, 2010.

[47] S. T. King and P. M. Chen, "Backtracking intrusions," in *SOSP*, 2003.

[48] G. Kurtz, "Operation Aurora Hit Google, Others," 2010, http://securityinnovator.com/index.php?articleID=42948&sectionID=25.

[49] K. J. Lang, B. A. Pearlmutter, and R. A. Price, *Results of the Abbadingo one DFA learning competition and a new evidence-driven state merging algorithm*. Springer Berlin Heidelberg, 1998.

[50] K. H. Lee, X. Zhang, and D. Xu, "High Accuracy Attack Provenance via Binary-based Execution Partition," in *NDSS*, 2013.

[51] K. H. Lee, X. Zhang, and D. Xu, "LogGC: garbage collecting audit log," in *CCS*, 2013.

[52] Y. Liu, M. Zhang, D. Li, K. Jee, Z. Li, Z. Wu, J. Rhee, and P. Mittal, "Towards a timely causality analysisfor enterprise security," in *NDSS*, 2018.

[53] 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*, 2017.

[54] S. Ma, X. Zhang, and D. Xu, "ProTracer: Towards Practical Provenance Tracing by Alternating Between Logging and Tainting," in *NDSS*, 2016.

[55] J. Mace, R. Roelke, and R. Fonseca, "Pivot tracing: Dynamic causal monitoring for distributed systems," in *SOSP*, 2015.

[56] P. Macko and M. Seltzer, "Provenance map orbiter: Interactive exploration of large provenance graphs." in *TaPP*, 2011.

[57] J. Morris, "svirt: Hardening linux virtualization with mandatory access control," in *Linux. conf. au Conference*, 2009.

[58] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer, "Provenance-aware Storage Systems," in *ATC*, 2006.

[59] D. Namiot and M. Sneps-Sneppe, "On micro-services architecture," *International Journal of Open Information Technologies*, vol. 2, no. 9, pp. 24–27, 2014.

[60] T. Pasquier, J. Singh, D. Eyers, and J. Bacon, "Camflow: Managed data-sharing for cloud services," in *IEEE Transactions on Cloud Computing*, 2015.

[61] D. Pohly, S. McLaughlin, P. McDaniel, and K. Butler, "Hi-Fi: Collecting High-Fidelity Whole-System Provenance," in *ACSAC*, 2012.

[62] K. Riesen, M. Neuhaus, and H. Bunke, *Bipartite Graph Matching for Computing the Edit Distance of Graphs*. Springer Berlin Heidelberg, 2007.

[63] R. Shu, X. Gu, and W. Enck, "A study of security vulnerabilities on docker hub," in *CODASPY*, 2017.

[64] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," Technical report, Google, Inc, Tech. Rep., 2010.

[65] B. Tak, C. Isci, S. Duri, N. Bila, S. Nadgowda, and J. Doran, "Understanding security implications of using containers in the cloud," in *USENIX ATC*, 2017.

[66] D. Wagner and P. Soto, "Mimicry attacks on host-based intrusion detection systems," in *CCS*, 2002.

[67] Q. Wang, W. U. Hassan, A. Bates, and C. Gunter, "Fear and logging in the internet of things," in *NDSS*, 2018.

[68] W. Wieczorek, *Grammatical Inference: Algorithms, Routines and Applications*. Springer, 2016, vol. 673.

[69] Y. Xie, D. Feng, Z. Tan, L. Chen, K.-K. Muniswamy-Reddy, Y. Li, and D. D. Long, "A Hybrid Approach for Efficient Provenance Storage," in *CIKM*, 2012.

[70] L. Zeng, Y. Xiao, and H. Chen, "Linux auditing: overhead and adaptation," in *IEEE ICC*, 2015.

[71] C. Zhao, J. Kong, and K. Zhang, "Program behavior discovery and verification: A graph grammar approach," *IEEE TSE*, vol. 36, no. 3, pp. 431–448, May 2010.

[72] W. Zhou, Q. Fei, A. Narayan, A. Haeberlen, B. T. Loo, and M. Sherr, "Secure Network Provenance," in *SOSP*, 2011.