Back-Propagating System Dependency Impact for Attack Investigation

Abstract

Causality analysis on system auditing data has emerged as an important solution for attack investigation. Given a POI (Point-Of-Interest) event (e.g., an alert fired on a suspicious file creation), causality analysis constructs a dependency graph, in which nodes represent system entities (e.g., processes and files) and edges represent dependencies among entities, to reveal the attack sequence. However, causality analysis often produces a huge graph ($> 100,000$ edges) that is hard for security analysts to inspect. From the dependency graphs of various attacks, we observe that (1) dependencies that are highly related to the POI event often exhibit a different set of properties (e.g., data flow and time) from the less-relevant dependencies; (2) the POI event is often related to a few attack entries (e.g., downloading a file). Based on these insights, we propose DEPIMPACT, a framework that identifies the critical component of a dependency graph (i.e., a subgraph) by (1) assigning *discriminative dependency weights* to edges to distinguish *critical edges that represent the attack sequence* from less-important dependencies, (2) propagating dependency impacts backward from the POI event to entry points, and (3) ranking entry points by their dependency impacts. In particular, DEPIMPACT performs forward causality analysis from the top-ranked entry points that are likely to be the attack entries to filter out edges in the original dependency graph that are not found in the forward causality analysis. Our evaluations on the 150 million real system auditing events of real attacks and the DARPA TC dataset show that DEPIMPACT can significantly reduce the large dependency graphs ($\sim 1,000,000$ edges) to a small graph (~ 160 edges), which is $\sim 6,250\times$ smaller. The comparison of DEPIMPACT with the other four state-of-the-art causality analysis techniques shows that DEPIMPACT is at least $106\times$ more effective in reducing the dependency graphs for revealing the attack sequences.

1 Introduction

Recent cyber attacks have plagued many well-protected businesses, causing significant financial losses [4–8, 20, 68]. These attacks often exploit multiple types of vulnerabilities to infiltrate into target systems in multiple stages, posing challenges for detection and investigation. To counter these attacks, recent approaches based on *ubiquitous system monitoring* have emerged as an important approach for monitoring system activities and performing attack investigation [24, 25, 37, 38, 41, 42, 48, 49]. System monitoring collects kernel auditing events about system calls as system audit logs. The collected data enables approaches based on *causality analysis* [27, 30, 38, 40–42, 48, 50] to identify entry points of intrusions (backward tracing) and ramifications of attacks (forward tracing), which have been shown to be effective in reducing false alerts of intrusions [30, 60, 64] and assisting timely system recovery [27, 40].

Despite the great promise of causality analysis, existing approaches require non-trivial efforts of inspection [30, 31], which limits their wide adoption. Causality analysis approaches assume causal dependencies between system entities (e.g., files, processes, and network connections) that are involved in the same system call event (e.g., a process reading a file). Based on such assumption, these approaches organize system call events in a system dependency graph, with nodes being system entities and edges being system events. By inspecting such a dependency graph, security analysts can *obtain the contextual information of an attack* by reconstructing the chain of events that leads to the POI (Point-Of-Interest) event (i.e., an alert event reported by anomaly detection tools or manually observed). Such contextual information is particularly effective in distinguishing benign and attack-related events such as distinguishing benign uses of *ZIP* from ransomware [30, 39]. However, due to the dependency explosion problem [46, 67, 71], the dependency graph could be gigantic, typically containing >100,000 edges [30, 31]. As a result, it is difficult for security analysts to soundly reason the graph, and find the edges that are critical to the attack.

Key Insight. By carefully inspecting the dependency graphs of various attacks [25, 38, 48, 50], we have two key observations. First, on a large dependency graph constructed from a POI event, a small number of *critical edges* (e.g., events that create and execute malicious payloads) that represent the attack sequence are typically buried in many non-critical edges (e.g., events that perform irrelevant system activities).

Compared to non-critical edges, critical edges typically exhibit a different set of properties and are more related to the POI event in these properties. For example, critical edges that read data from a suspicious IP and then write the data to a malicious script file will have the similar data flow amount as the script file's size. Second, a POI event is often caused by a few sources, referred to as *attack entries*. These attack entries are represented as entry points of the attack sequence that lead to the POI event, and are buried in many other irrelevant entry nodes (i.e., nodes without incoming edges) in the dependency graph. For example, many attacks start by injecting a malicious script into the victim host and may further download more tools along the attack. Such an attack is captured in a dependency graph with the attack entries representing the downloaded malicious script and tools.

Challenges. While identifying critical edges and attack entries has the great potentials in reducing the size of the dependency graph while preserving the attack sequence, there are three major challenges for achieving such goals.

First, the processes that are causally related to the POI event usually perform other irrelevant system activities in the background, causing a large number of less-important dependencies to be included in the dependency graphs. Moreover, these irrelevant system activities often trace back to many irrelevant sources (e.g., irrelevant web browsing and file downloads) that have low impact on the POI event, and thus causality analysis may identify more than a thousand entry nodes (Section 5.1). As a result, it is often infeasible to manually inspecting these daunting number of edges and entry nodes to identify critical edges and attack entries.

Second, data flow amount seems like a promising feature for distinguishing critical edges in some attacks. However, based on our empirical observations (Section 5.1), for many attacks, there are usually lots of non-critical edges that have the similar data amount as the critical edges in the dependency graphs. This indicates that a single feature is limited in addressing diversified attack scenarios.

Third, while existing techniques have also made attempts to identify critical edges, they mainly rely on heuristic rules that cause loss of information [41], intrusive system changes [38, 50] such as binary instrumentation and kernel customization, or execution profiles [30], hindering their practical adoption. We need a general solution that does not suffer from the same adoption limitations as these existing techniques.

Contributions. Based on the key insights, we propose DEPIMPACT, a framework that *facilitates attack investigation by identifying critical edges and attack entries in large dependency graphs, without heuristic rules, intrusive system changes, or execution profiles*. Specifically, given a POI event to be investigated, DEPIMPACT first applies causality analysis to construct a backward dependency graph for the POI event, and then employs automated techniques to identify the *critical component* of the dependency graph. Critical component, by definition, is a subgraph of the dependency graph that preserves the information critical to attack investigation (i.e., critical edges and attack entries). Compared to the original dependency graph, the size of the critical component is significantly reduced, which drastically reduces the complexity for obtaining the contextual information of the attack.

In its design, DEPIMPACT develops three major techniques to address the aforementioned technical challenges.

(1) Dependency Weight Computation: Instead of using timing analysis only, DEPIMPACT captures the differences between critical edges and non-critical edges by profiling multiple features for each edge, including timing, data flow amount, and node degree (Section 4.2.2). Then, DEPIMPACT employs a *discriminative feature projection scheme* based on Linear Discriminant Analysis (LDA) [52] to compute a weight score based on the features, referred to as *dependency weight* (Section 4.2.3). This scheme aims to maximize the weight differences between critical and non-critical edges. The dependency weight of an edge, ranging from 0.0 to 1.0, quantifies the correlation between the edge and the POI event. An edge with a higher dependency weight implies more relevance to the POI event, and is more likely to be a critical edge.

(2) Dependency Impact Back-Propagation & Entry Node Ranking: To reveal attack entries, DEPIMPACT employs a notion of *dependency impact*. The dependency impact of a node is defined as a score that models the node's impact on the POI event, i.e., a higher score implies a higher impact. To compute the dependency impacts for all nodes, DEPIMPACT employs a weighted score propagation scheme that propagates the dependency impact from the nodes in the POI event backward along the edges to all entry nodes. Inspired by TrustRank [29], our score propagation scheme computes the dependency impact of a node as a weighted sum of its children's dependency impact scores where each child node's weight is the normalized dependency weight of the edge between the parent node and the child node. The intuition behind our score propagation scheme is that an attack entry's impact on the POI event is proportionally distributed to its children based on the edge dependency weights. After propagation, DEPIMPACT ranks the entry nodes based on their dependency impacts, and the top-ranked entry nodes are more likely to be attack entries.

(3) Forward Causality Analysis for Critical Component Identification: After ranking the entry nodes, DEPIMPACT performs forward causality analysis from the top-ranked entry nodes, producing another dependency graph, called *forward dependency graph*. The overlapping part between the forward graph and the original backward dependency graph accurately preserves the nodes and edges that are highly relevant to both the POI event and the attack entries. We refer to this overlapping part as the *critical component* of the original dependency graph.

Evaluation. We implemented a prototype of DEPIMPACT in roughly $\sim$20K lines of code and deployed it on a physical testbed to collect real system auditing data and perform attack investigation. We performed 7 attacks that are used in prior studies [19,45,48,71] and 3 multi-host intrusive attacks based on the Cyber Kill Chain framework [3] and CVE [55], and applied DEPIMPACT to investigate them. During our evaluation, the deployed hosts continue to resume their routine tasks to emulate the real-world deployment where irrelevant sys-

tem activities and attack activities co-exist. We additionally include 5 attack cases in DARPA TC dataset [18] in our evaluation. In total, we collected ∼100 million system auditing events for our performed attacks and the DARTA TC dataset contains ∼50 million events. Our tool and dataset is available at *https://github.com/usenixsub/DepImpact*.

The evaluation results demonstrate that DEPIMPACT is highly effective in revealing critical edges and attack entries. On average, the size of the critical component produced by DEPIMPACT has ∼ 160 edges, which is ∼ 6250× smaller than the size of the original dependency graph (∼ 1 million edges). Such a high reduction rate is achieved *without missing any critical edge*, which is mainly due to the fact that DEPIMPACT consistently *ranks the attack entries at the top*, demonstrating the effectiveness in using the top-ranked entry nodes for identifying critical components. The comparison with four other state-of-the-art causality analysis techniques (CPR [71], ReadOnly [47], PrioTracker [48], and NoDoze [30]) shows that DEPIMPACT is *at least* 106× *more effective in dependency graph reduction*. Additionally, compared with the version of DEPIMPACT that uses less features and the average-projection approach that uses an average projection vector for computing dependency impacts, DEPIMPACT achieves at least 69.91% improvement in ranking attack entries, demonstrating the superiority of DEPIMPACT's discriminative feature projection scheme and proving the necessity of features. Finally, DEPIMPACT finishes analyzing an attack within 6 minutes, which is ∼ 4× faster when compared with the average-projection approach. The results also show that DEPIMPACT and the state-of-the-art technique, NoDoze, have similar runtime performance for most of the attacks, while DEPIMPACT achieves much better reduction rate than NoDoze.

2 Background and Motivation

2.1 System Monitoring

System monitoring collects auditing events about system calls that are crucial in security analysis, describing the interactions among system entities. As shown in previous studies [24, 25, 27, 30, 37, 38, 41, 42, 48, 49], on mainstream operating systems (Windows, Linux, and Mac OS), system entities in most cases are files, processes, and network connections, and the collected system calls are mapped to three major types of system events: (1) file access, (2) processes creation and destruction, and (3) network access. Following the established trend, in this work, we consider *system entities* as *files*, *processes*, and *network connections*. We consider a *system event* as the interaction between two system entities represented as ⟨*subject, operation, object*⟩. Subjects are processes originating from software applications (e.g., Chrome), and objects can be files, processes, and network connections. We categorize system events into three types according to the types of their object entities, namely *file events*, *process events*, and *network events*. Both entities and events have critical security-related attributes (Tables 1 and 2). Representative attributes of entities include file name, process executable name, IP, and port. Representative attributes of events include event origins (e.g., start time/end time) and operations (e.g., file read/write).

Table 1: Representative attributes of system entities

Entity	Attributes	Shape in Graph
File	Name, Path	Ellipse
Process	PID, Name, User, Cmd	Square
Network Connection	IP, Port, Protocol	Parallelogram

Table 2: Representative attributes of system events

Operation	Read/Write, Execute, Start/End
Time	Start Time/End Time, Duration
Misc.	Subject ID, Object ID, Data Amount, Failure Code

2.2 Causality Analysis

Causality analysis [27, 30, 38, 40–42, 48, 50] analyzes the auditing events to infer their dependencies and present the dependencies as a directed graph. In the dependency graph $G(E,V)$, a node $v \in V$ represents a process, a file, or a network connection. An edge $e(u,v) \in E$ indicates a system auditing event that involves two entities u and v (e.g., process creation, file read or write, and network access), and its direction (from the source node u to the sink node v) indicates the direction of data flow. Each edge is associated with a time window, $tw(e)$. We use $ts(e)$ and $te(e)$ to represent the start time and the end time of e. Formally, in the dependency graph, for two events $e_1(u_1,v_1)$ and $e_2(u_2,v_2)$, there exists causal dependency between e_1 and e_2 if $v_1 = u_2$ and $ts(e_1) < te(e_2)$.

Causality analysis enables two important security applications: (1) *backward causality analysis* that identifies entry points of attacks, and (2) *forward causality analysis* that investigates ramifications of attacks. Given a POI event $e_s(u,v)$, a backward causality analysis traces back from the source node u to find all events that have causal dependencies on u, and a forward causality analysis traces forward from the sink node v to find all events on which v has causal dependencies.

2.3 Motivating Example

Figure 1 shows a partial dependency graph of a file hiding activity: a suspicious script `mal.sh` is executed to download a malicious file `mal` from a remote host `192.1.1.254`. The file is then moved to `user/mal` and renamed to `user/file.txt`. Given a POI event which renames the file to `user/file.txt`, the dependency graph produced by backward causality analysis contains 194,208 nodes and 3,273,769 edges. The critical edges and attack entries (`192.1.1.254`, `mal.sh`) that represent the attack sequence are colored in dark black. The goal of attack investigation is to inspect the dependency graph to reveal critical edges and attack entries of the attack.

Challenges. As observed in Figure 1, attack investigation is a process of *finding a needle in a haystack*: a limited number of critical edges (i.e., 12) are buried in an overwhelmingly large number (∼ 3 million) of non-critical edges (i.e., less-important dependencies), and same for attack entries (i.e., 2 out of ∼ 35K irrelevant entry nodes).

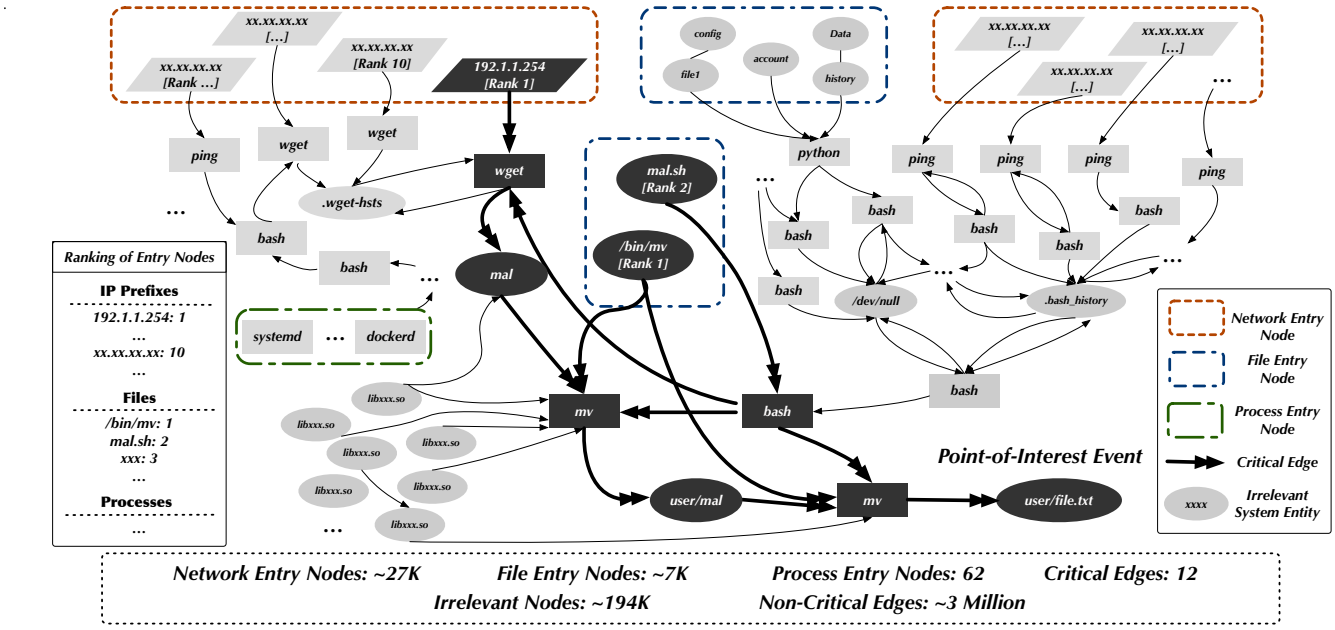

Figure 1: Partial dependency graph of an attack that downloads a malicious file and hides the file by renaming it (rectangles for processes, ovals for files, parallelograms for network connections). The complete dependency graph constructed from the POI event (renaming to `user/file.txt`) via backward causality analysis contains 194,208 **nodes and** 3,273,769 **edges. The critical component identified by DEPIMPACT is colored in dark black, which contains** 10 **nodes (including** 2 **attack entries) and** 12 **edges (these edges are all critical edges). As can be seen, DEPIMPACT significant reduces the size of the dependency graph while preserving the critical attack information.**

Using DEPIMPACT to Identify Critical Component. DEPIMPACT first divides the entry nodes into 3 categories (i.e., network connections, files, and processes), and ranks the entry nodes in each category. Here, DEPIMPACT ranks the IP `192.1.1.254` for `mal` downloading as top 1, the malicious script `mal.sh` and the executable `/bin/mv` as top 1 and top 2. By performing forward causality analysis from top-ranked entry nodes and taking the overlap, DEPIMPACT filters out most less-important dependencies ($\sim$ 3 million) and identifies the critical component (colored in dark black; 10 nodes, 12 edges) that preserves all critical edges and attack entries.

3 Overview

Figure 2 shows the architecture of DEPIMPACT. Given a POI event, DEPIMPACT automatically identifies the critical component of the dependency graph produced by causality analysis. DEPIMPACT consists of three phases: (1) dependency graph generation, (2) dependency weight computation, and (3) critical component identification.

In Phase I, DEPIMPACT leverages mature system auditing frameworks [15, 51, 62, 66] to collect system audit logs. Given a POI event, DEPIMPACT parses the collected logs and performs backward causality analysis [41, 42] to generate a backward dependency graph for the POI event. In Phase II, DEPIMPACT first employs state-of-the-art dependency graph reduction techniques [71] to reduce the graph size (Section 4.2.1). Then, DEPIMPACT extracts features for edges and employs a discriminative feature projection scheme based on LDA to compute dependency weights from the features, so that critical edges can be better revealed. The output of Phase II is a weighted dependency graph for the POI event. In Phase III, DEPIMPACT first employs a weighted score propagation scheme to propagate the dependency impact from the POI event backward along the edges to all entry nodes. Then, DEPIMPACT ranks entry nodes based on their dependency impacts and selects the top candidates. Finally, DEPIMPACT performs forward causality analysis from the top-ranked entry nodes and identifies the overlap of the backward dependency graph and the forward dependency graph as the critical component for output.

Threat Model. Our threat model is similar to the threat model of previous work on system monitoring [24, 25, 30, 41, 42, 47, 48]. We assume that kernel and kernel-layer auditing framework [15, 51, 62, 66] are part of our trusted computing base (TCB), and existing software and kernel hardening techniques [12, 17] can be used to secure log storage. Any kernel-level attack that deliberately compromises security auditing systems is beyond the scope of this work. We assume an outside attacker that attacks the system remotely (from outside of the system). Thus, the attacker either utilizes the vulnerabilities in the system or convinces the user to download a file with malicious payload.

We do not consider the attacks performed using implicit flows (e.g., side channels) or inter-procedural communications (IPC) that do not go through kernel-layer auditing and thus cannot be captured by the underlying provenance tracker. Finer-grained auditing tools that capture memory traces or

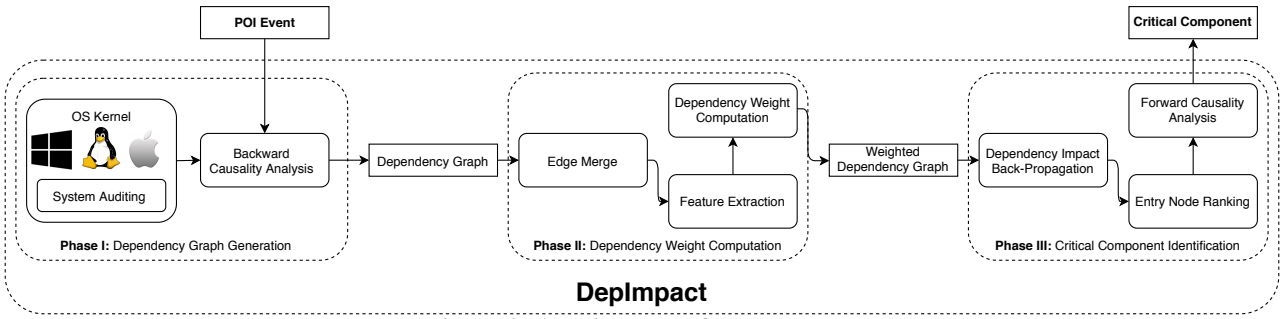

Figure 2: Architecture of DEPIMPACT

Table 3: Representative system calls processed

Event Category	Relevant System Call
Process/File	read, write, readv, writev
Process/Process	execve, fork, clone
Process/Network	read, write, sendto, recvfrom, recvmsg

program analysis techniques can be used to address these types of the attacks and it is not the focus of this work. We also do not consider Mimicry attacks [69] where attackers deliberately evade intrusion detection systems through a chain of events that seem benign in enterprises. Existing intrusion detection systems [13, 44, 56] often rely on heuristics or analysis based on the properties of a single event, and thus are vulnerable to such attacks. While detecting Mimicry attacks is the limitation of the detection systems, it is beyond the scope of this work since our focus is to identify the relevant events as the contextual information for the alerts generated by the detection systems.

4 Design of DEPIMPACT

4.1 Dependency Graph Generation

4.1.1 System Auditing

DEPIMPACT leverages mature system auditing frameworks [15,51,62,66] to collect system audit logs about system calls from the kernel. DEPIMPACT then parses the collected logs to build a *global* system dependency graph, where nodes represent system entities and edges represent system (call) events. In particular, DEPIMPACT focuses on three types of system entities/events: (i) file access, (ii) process creation and destruction, and (iii) network access. Table 3 shows the representative system calls (in Linux) processed by DEPIMPACT. Failed system calls are filtered out by DEPIMPACT, as processing them will cause false dependencies among events. Tables 1 and 2 show the representative attributes of entities and events extracted by DEPIMPACT. Following the existing work [24, 25, 48], to uniquely identify entities, for a process entity, we use the process name and PID as its unique identifier. For a file entity, we use the absolute path as its unique identifier. For a network connection entity, we use 5-tuple ($\langle$*srcip, srcport, dstip, dstport, protocol*$\rangle$) as a network connection's unique identifier. Failing to distinguish different entities causes problems in relating events to entities and tracking the dependencies among events.

4.1.2 Backward Causality Analysis

Given a POI event, DEPIMPACT performs backward causality analysis (Section 2.2) to generate a *local* backward dependency graph G_d for the POI event. Briefly speaking, backward causality analysis adds the POI event to a queue, and repeats the process of finding eligible incoming edges of the edges/events (i.e., incoming edges of the source nodes of edges) in the queue until the queue is empty. The output of Phase I is a backward dependency graph that only contains system events (and associated entities) and that are causally dependent on the POI event.

4.2 Dependency Weight Computation

4.2.1 Edge Merge

The dependency graph produced by causality analysis often has many parallel edges between two nodes [71]. The reason is that OS typically finishes a read/write task (e.g., file read/write) by distributing the data proportionally to multiple system calls. Inspired by the recent work for dependency graph reduction [71], DEPIMPACT merges the edges between two nodes if the time differences of these edges are smaller than a given threshold. We tried different values for the merge threshold and selected 10s, as it gives reasonable results in merging system calls for file manipulations, file transfers, and network communications, which is consistent with [71].

4.2.2 Feature Extraction

For each edge, DEPIMPACT extracts three features to compute a dependency weight, which models the correlation between the edge and the POI event. An edge with a higher dependency weight implies more relevance to the POI event, and is more likely to be a critical edge.

Data Flow Relevance $f_{S(e)}$. Intuitively, edges that have similar data flow amount as the data size of the entities in the POI event are more likely to be relevant. As such, we design feature $f_{D(e)}$ to model the data flow relevance of an edge $e(u,v)$ to the POI event:

$$f_{S(e)} = 1/(\mid s_e - s_{e_s} \mid + \alpha) \quad (1)$$

where s_e and s_{e_s} represent the data flow amount associated with the edge e and the POI event e_s. The smaller the difference $\mid s_e - s_{e_s} \mid$, the higher the data flow relevance $f_{S(e)}$. Note

that we use a small positive number α (we set $\alpha = 1e{-}4$) to handle the special case when e is the POI event: POI event has the highest feature value $f_{S(e_s)} = 1/\alpha$.

Temporal Relevance $f_{T(e)}$. Intuitively, edges that occurred at relatively the same time are more likely to be relevant. As such, we design feature $f_{T(e)}$ to model the temporal relevance of an edge $e(u,v)$ to the POI event:

$$f_{T(e)} = \ln(1 + 1/\mid t_e - t_{e_s} \mid) \tag{2}$$

where $t_{(e)}$ and t_{e_s} represent the timestamp values (we use the event end time) of the edge e and the POI event e_s. The smaller the difference $\mid t_e - t_{e_s} \mid$, the higher the temporal relevance $f_{T(e)}$. To handle the special case when e is the POI event (i.e., $\mid t_e - t_{e_s} \mid = 0$), we use one tenth of the minimal time unit (nanosecond) in the audit logging framework (i.e., 1e-10) to compute its feature value: $f_{T(e_s)} = ln(1 + 1e10)$. This ensures that the POI event has the highest feature value.

Concentration Ratio $f_{C(e)}$. In the backward causality analysis, if the number of source nodes that can be traced from a node v is 1 (i.e., only one incoming edge from v), we say that the dependency represented by this edge is highly concentrated for v. Also, we would like to give higher weights to the node that can be reached from multiple paths in the backward direction. Thus, we define the *concentration ratio* for the edge $e(u,v)$ as:

$$f_{C(e)} = OutDegree(v)/InDegree(v) \tag{3}$$

Here, $InDegree(v)$ and $OutDegree(v)$ represent the in-degree and out-degree of the sink node v.

4.2.3 Dependency Weight Computation

To compute a dependency weight from the features, DEPIMPACT leverages linear projection that is known for high interpretability and low computational cost [22]. Instead of directly taking the average, DEPIMPACT employs a *discriminative feature projection scheme* based on Linear Discriminant Analysis (LDA) [52] to compute a projection vector to maximize the differences between critical edges and non-critical edges, with critical edges assigned with higher weights. Next, we present the scheme in detail.

Step 1: Edge Clustering. In the first step, DEPIMPACT leverages clustering to separate edges into two groups: one is likely to contain critical edges, and the other for non-critical edges. Specifically, DEPIMPACT first normalizes features to 0-1 range [22], and then employs Multi-KMeans++ clustering algorithm [11]. KMeans clustering algorithm aims to partition the points into k clusters such that each point belongs to the cluster with the nearest center. Based upon KMeans, KMeans++ improves the initial seeds selection to avoid poor clustering. Multi-KMeans++ is a meta algorithm that performs n runs of KMeans++ and then chooses the best clustering that has the lowest distance variance over all clusters. We choose $k = 2$ since we want to cluster edges into two groups, as required by LDA. We experimented a range of values for n ($[5, 30]$) and chose $n = 20$ as it delivers the best clustering results without much overhead.

Step 2: Discriminative Feature Projection. Given two groups of edges, DEPIMPACT leverages Linear Discriminant Analysis (LDA) [52] to compute an optimal projection vector that maximizes the separation between group projections. LDA finds the optimal projection plane such that the projected points in the same group are close to each other, and the projected points in different groups are far from each other. Formally, LDA finds the projection vector ω that maximizes the Fisher criterion, $J(\omega) = \frac{\omega^T S_b \omega}{\omega^T S_w \omega}$, where S_b and S_w are between-group scatter matrix and within-group scatter matrix, respectively. Solving the optimization problem yields:

$$\omega^* = \arg\max J(\omega) = S_w^{-1}(\mu_1 - \mu_2) \tag{4}$$

Denote the solution to Equation (4) as $\omega^* = [\omega_S^* \; \omega_T^* \; \omega_C^*]^T$. For an edge e, its unnormalized weight $W_{e_{UN}}$ is computed as:

$$W_{e_{UN}} = \omega_S^* f_{S(e)} + \omega_T^* f_{T(e)} + \omega_C^* f_{C(e)} \tag{5}$$

One remaining issue is that Equation (4) does not guarantee the direction of the projection vector, and it might be possible that critical edges have lower weights than non-critical edges. To address the issue, we leverage the observation that, in most cases, the number of critical edges is significantly less than the number of non-critical edges (as can be seen from attack cases in Section 5.1). Specifically, we negate the direction of the projection vector if the average of the projected weights for a smaller edge group (likely to be the group of critical edges) is smaller. As shown in Section 5.4, compared to the naive approach of taking the average of features (the average-projection approach), our feature projection scheme preserves as much of the group discriminatory information as possible and leads to better performance for entry node ranking.

Step 3: Edge Weight Normalization. For an edge $e(u,v)$, we normalize its projected weight by the sum of weights of all outgoing edges of the source node u:

$$W_e = W_{e_{UN}} / \sum_{e' \in outgoingEdge(u)} W_{e'_{UN}} \tag{6}$$

The rationale behind is to ensure that for each node, the weights of all its outgoing edges are in the range $[0.0, 1.0]$ and the sum of the weights is equal to 1.0. Coupled with our score propagation scheme for dependency impact (Section 4.3), such way of normalization ensures that (1) the dependency impact of any node does not exceed the maximum dependency impact of its child nodes, and (2) the dependency impact of any node does not exceed the dependency impact of the nodes in the POI event (i.e., 1.0). The output of Phase II is a weighted backward dependency graph for the POI event, in which the dependency weights encode the differences between critical edges and non-critical edges.

Algorithm 1: Dependency Impact Propagation

Input: Weighted dependency graph, G
threshold, δ
Output: Weighted dependency graph, G; nodes are associated with dependency impact scores

1 $POI.score \leftarrow 1$
2 **while** $diff > \delta$ **do**
3 $\quad diff \leftarrow 0$
4 $\quad$ **for** $\forall u \in G$ **do**
5 $\quad\quad$ **if** u *is POI* **then**
6 $\quad\quad\quad$ **continue**
7 $\quad\quad$ **else**
8 $\quad\quad\quad res \leftarrow 0$
9 $\quad\quad\quad$ **for** $\forall v \in G.childNodes(u)$ **do**
10 $\quad\quad\quad\quad res \mathrel{+}= v.score * G.edge(u,v).weight$
11 $\quad\quad\quad diff \mathrel{+}= |u.score - res|$
12 $\quad\quad\quad u.score \leftarrow res$

4.3 Critical Component Identification

4.3.1 Dependency Impact Back-Propagation

Given a weighted dependency graph, DEPIMPACT propagates the dependency impact from the POI event to all other nodes backward along the weighted edges. The dependency impact for the nodes (both source node and sink node) in the POI event is 1.0 by default. For a node u, its dependency impact is iteratively updated by taking the weighted sum of dependency impacts of its child nodes:

$$DI_u = \sum_{v \in childNodes(u)} DI_v * W_{e(u,v)} \tag{7}$$

where DI_u denotes the dependency impact of node u and $W_{e(u,v)}$ denotes the dependency weight (after normalization) of edge $e(u,v)$. Such score propagation scheme guarantees that the score of any node does not exceed the maximum score of its child nodes, and the score of any node does not exceed the score of the nodes in the POI event. Furthermore, compared to the distribution-based score propagation algorithms like PageRank [57], our scheme preserves the scores along long dependency paths and prevents fast degradation.

Algorithm 1 illustrates our dependency impact score propagation algorithm. In each iteration, the algorithm updates the dependency impact score of each node by taking the weighted sum of the scores of all its child nodes (Line 10), and computes the sum of score differences for all nodes (Line 11). The propagation terminates when the aggregate difference between the current iteration and the previous iteration is smaller than a threshold, δ (Line 2), indicating that the scores of all nodes have reached a stable point. We set $\delta = 1e\text{-}13$ as it gives robust results from our evaluations.

4.3.2 Entry Node Ranking

After dependency impact propagation, DEPIMPACT ranks the entry nodes based on their dependency impacts. The intuition behind entry node ranking is that entry nodes with higher dependency impacts are more related to the POI event and are more likely to be the attack entries, and thus their descendant nodes and associated edges are more likely to be included in the critical component.

In the current design, we have a special treatment of system library nodes. As has been shown in prior work [67], system library files are typically loaded by certain processes, and do not have incoming edges on the dependency graph. As the number of system library nodes could be potentially large, naively treating them all as entry nodes could add significant difficulties to entry node ranking and attack entry candidates selection, impairing the final results. Thus, for system library nodes, we take the process nodes that load them as entry nodes. Specifically, we classify entry nodes into three categories: (1) file entry node: file nodes that do not have incoming edges except system libraries; (2) process entry node: process nodes whose parent nodes are all system libraries; (3) network entry node: network nodes that do not have incoming edges. We then select the top-ranked entry nodes from each category.

4.3.3 Critical Component Identification

From the top-ranked entry nodes, DEPIMPACT performs forward causality analysis until reaching the POI event. As a final step, DEPIMPACT identifies the overlap of the backward dependency graph and the forward dependency graph as the critical component for output. Compared to the original large backward dependency graph, the critical component contains the parts of dependencies that are actually relevant to the POI event and its size is significantly reduced. Furthermore, the critical component illustrates how the attack-relevant information flows from attack entries to the POI event through critical edges, which facilitates further attack investigation.

5 Evaluation

We built DEPIMPACT (∼20K lines of code in Java) upon Sysdig [66], and evaluate DEPIMPACT using both the attack cases constructed based on the known exploits [19, 45, 48, 71] and the attack cases collected by the DARPA Transparent Computing (TC) program [18]. In the evaluations, we aim to answer the following research questions:

- **RQ1**: How effective is DEPIMPACT in revealing attack sequences in comparison with other state-of-art techniques?
- **RQ2**: How many top-ranked entry nodes should be used in DEPIMPACT for revealing attack sequences?
- **RQ3**: How effective is DEPIMPACT in revealing attack entries?
- **RQ4**: How efficient is DEPIMPACT in investigating an attack?

5.1 Evaluation Setup

We deployed Sysdig [66] on 5 Linux hosts to collect system auditing events and then applied DEPIMPACT to perform attack investigation. DEPIMPACT is executed on a server with

Table 4: Statistics of dependency graphs generated for all the 15 attacks

Attack	Causality Analysis # V	Causality Analysis # E	Edge Merge #V	Edge Merge # E	Entry Nodes	Critical Edge	Attack Entries	POI
Wget Executable	126	673	126	363	46	8	2	∼50MB
Illegal Storage	8,450	93,085	8,450	62,073	960	6	2	∼50MB
Illegal Storage2	42,450	658,913	42,450	378,326	3,499	4	2	∼50MB
Hide File	194,208	6,464,098	194,208	3,273,769	35,203	12	2	∼50MB
Steal Information	195,636	6,493,626	195,636	3,291,208	35,213	4	2	∼50MB
Backdoor Download	7,510	69,479	7,510	60,390	157	8	2	∼50MB
Annoying Server User	114	585	114	318	34	10	2	∼50MB
Shellshcok	1,648	20,332	1,648	3,600	1,273	30	3	124B
Dataleak	407	2,262	407	1,152	234	18	3	7.1KB
VPN Filter	1,195	5,212	1,195	1,879	999	10	2	1.6KB
Five Dir Case1	240	272	240	272	232	2	1	50.78KB
Five Dir Case3	5,907	78,075	5,907	78,075	879	4	1	121.85KB
Theia Case1	184,352	816,277	184,352	816,277	151,827	8	2	166.78KB
Theia Case3	334,441	1,500,717	334,441	1,500,717	282,651	6	2	166.64KB
Trace Case5	263	971	263	971	28	3	1	95.KB
AVG	65,129.80	1,080,305.13	65,129.80	631,292.67	34,215.67	8.87	1.93	–

an Intel(R) Xeon(R) CPU E5-2637 v4 (3.50GHz), 256GB RAM running 64bit Ubuntu 18.04.1. For investigating the attack cases based on the known exploits, we performed 10 attacks in the deployed environment: 7 attacks based on commonly used exploits and 3 multi-host and mutli-step intrusive attacks based on the Cyber Kill Chain framework [3] and CVE reports [55]. The deployed hosts have 12 active users with hundreds of processes, and are used for various types of daily tasks such as file manipulation, text editing, and software development, which are representative of real-world usage. During evaluation, the deployed hosts continue to resume their routine tasks to emulate the real-world deployment where irrelevant system activities and attack activities co-exist. The routine tasks on these machines ensure that enough noise of irrelevant system activities is collected. In total, the real system audit logs collected in our deployed hosts contain *∼100 million* events. The DARPA dataset includes system audit logs collected from 5 hosts with different OS systems. We developed a tool to parse the released logs and loaded the events into our databases. In total, the DARPA dataset used in our evaluation contains *∼50 million* events. We next describe these attacks in detail.

5.1.1 Attacks Based on Commonly Used Exploits

These 7 attacks are used in prior work's evaluations [19, 45, 48, 71], which consist of the following scenarios:

- *Wget Executable*: A vulnerable server allows the attacker to download executable files using wget. The attacker downloads python scripts and executes the scripts.
- *Illegal Storage*: A server administrator uses wget to download suspicious files to a user's home directory.
- *Illegal Storage 2*: A server administrator uses curl to download suspicious files to a user's home directory.
- *Hide File*: The goal of the attacker is to hide malicious file among the user's normal files. The attacker downloads the malicious script and hides it by changing its file name and location.
- *Steal Information*: The attacker steals the user's sensitive information and writes the information to a hidden file.
- *Backdoor Download*: A malicious insider uses the ping command to connect to the malicious server, and then downloads the backdoor script from the server and hides the script by renaming it.
- *Annoying Server User*: The annoying user logs into other user's home directories on a vulnerable server and writes some garbage data to other user's files.

5.1.2 Multi-host Intrusive Attacks

These 3 multi-host intrusive attacks capture the important traits of attacks depicted in the Cyber Kill Chain framework [3] and CVE [55]. In these 3 attacks, the attacker uses an external host, referred to as the C2 (Command and Control) server, to perform penetration, distribute malware, and receive data. The first host that is compromised by the attack is called Host 1, which is a starting point to perform lateral movement and other malicious actions to compromise more hosts inside the network (i.e., Host 2, . . . , Host n).

Attack 1: Shellshock Penetration. After the initial shellshock penetration at Host 1, the attacker connects to Cloud services (e.g., Dropbox, Twitter) and downloads an image where C2 server's IP address is encoded in the EXIF metadata. The behavior is a common practice shared by APT attacks [10, 21] to evade the network-based detection system based on DNS blacklisting. Based on the IP, the attacker downloads a malware from the C2 server to Host 1. When the script is executed, it scans the ssh configuration file to locate reachable hosts in the network, discovering Host 2, Host 3, and Host 4. After this discovery phase, the malware downloads another script from the C2 server and sends it to these discovered hosts and steals password from them.

Attack 2: Data Leakage After Shellshock Penetration. After the previous reconnaissance, the attacker downloads another malware, `leak_data.sh`, from the C2 server and sends it to Host 2. The malware scans for hidden files and files containing sensitive strings, and compresses them in a tarball `leak.tar.bz2`. The malware then transfers the tarball back to Host 1. On Host 1, the tarball is encrypted and uploaded to the Internet.

Attack 3: VPN Filter. At this stage, the attacker seeks to maintain a direct connection to the victim hosts from the

C2 server. He utilizes the notorious VPN Filter malware [9] which infected millions of IoT devices by exploiting a number of known or zero-day vulnerabilities [1, 2]. After the initial penetration on Host 1 and discovery of Host 2, the attacker downloads the VPN Filter stage 1 malware from the C2 server to Host 1 and transfers it to Host 2. This malware then downloads another executable from the C2 server, and executes it to launch the attack and establish a connection to the C2 server. Using this connection, the attacker transfers a malicious script to Host 2 which will gather sensitive data on Host 2.

5.1.3 DARPA TC Attack Cases

The dataset released by the DARPA TC program contains attack cases performed on different operating systems. Based on the attack descriptions provided in the dataset, we exclude the attack cases that fail to launch the attacks, and the attack cases on the Android system since mobile applications' behaviors are constrained by the Android sandbox and are not suitable for our analysis. We also exclude the phishing e-mail attacks since most of their operations are through clicking links in the browsers and leave limited traces in the system audit logs. In total, we chosen five attacks that target at different operating systems (Linux, Windows) and exploit different vulnerabilities (Firefox backdoor and browser extensions).

5.1.4 Obtaining Ground Truth for the Attacks

For the attack cases performed on our hosts, we identified the POI events based on the performed attacks and applied backward causality analysis from the POI events to obtain the system dependency graphs. For the attack cases in the DARPA dataset, we queried the databases that are loaded with the logs to identify the POI events based on the attack description, and applied backward causality analysis from the POI events to obtain the system dependency graphs. For the attacks involving multiple hosts, DEPIMPACT performs cross-host causality analysis based on the existing techniques [42, 48], which produces causality graphs that include special network connection edges to represent connections among multiple hosts. Finally, we manually identified the critical edges and the attack entries based on the knowledge of the performed attacks and the attack descriptions in these system dependency graphs.

Table 4 shows the statistics of the generated dependency graphs for the attacks. Columns "Causality Ana. # V" and "Causality Ana. # E" show the number of nodes and edges after performing the causality analysis from the POI events. Columns "Edge Mer. # V" and "Edge Mer. # E" show the number of nodes and edges after applying edge merges (Section 4.2.1). Columns "Entry Nodes" and "Critical Edge" show the number of entry nodes and critical edges of the dependency graphs. Column "Attack Entries" shows the number of entry nodes that are labelled as attack entries. Column "POI" shows the data size of the files in the POI events. We clearly observe that even after edge merges, there still remains a large number of edges in the dependency graphs (631K on average with the max being 3.3 million edges), which motivates the further pruning provided by DEPIMPACT. Moreover, in these 15 attacks, the files in the POI events have diversified sizes, ranging from 124 bytes to $50M$ bytes, and on average, there are $42,757$ edges (with the max being $962,706$) that have similar data sizes as the files in the POI events. Thus, directly using the data flow amount to reveal attack sequences will include lots of irrelevant edges in the results, which motivates DEPIMPACT to combine multiple features for computing edge weights to achieve better performance.

Table 5: Statistics of dependency graphs generated for the 10 attacks

Attack	CPR	ReadOnly	PrioTracker	NoDoze	DEPIMPACT
Wget Executable	363	58	58	288	48
Illegal Storage	62,073	16,211	6,948	10,260	43
Illegal Storage2	378,326	89,779	37,112	19,512	7
Hide File	3,273,769	613,303	114,614	37,251	437
Steal Information	3,291,208	618,025	115,223	20,426	750
Backdoor Download	60,390	15,990	6,024	269	20
Annoying Server User	318	56	39	227	23
Shellshcok	3,600	590	518	911	451
Dataleak	1,152	231	211	687	223
VPN Filter	1,879	298	244	217	263
Five Dir. Case1	272	18	18	257	7
Five Dir. Case3	78,075	77,824	7,496	598	33
Theia Case1	816,277	325,459	176,800	151,240	62
Theia Case3	1,500,717	537,424	269,277	9,015	10
Trace Case5	971	910	459	510	4
AVG	631,292.67	153,078.40	49,002.73	16,777.87	158.73

5.2 RQ1: Revealing Attack Sequences

To demonstrate the effectiveness of DEPIMPACT in revealing the attack sequence by pruning non-critical edges, we compare DEPIMPACT with 4 state-of-the-art techniques: CPR [71], ReadOnly [47], PrioTracker [48], and NoDoze [30]. DEPIMPACT uses 6 entry nodes, composed of the top 2 entry nodes from the 3 types of system entities (i.e., files, processes, and network connections), to perform forward causality analysis, which is shown to preserve all the critical edges (see Section 5.3). For the attacks involving multiple POI events, we applied DEPIMPACT on each of the POI events and then union the generated critical components. CPR merges edges between two nodes if the time differences between the edges are within a threshold (i.e., 10 seconds). ReadOnly removes the edge whose source node is the read-only file. PrioTracker mainly uses the fanout of nodes to prioritize the dependencies in the causality analysis. We then adapt the computed priories as the dependency weights for edges and filter the edges with low weights. NoDoze assigns an anomaly score for each edge based on the frequency of the corresponding system event, and then computes the anomaly score for each path. As NoDoze requires an execute profile, we use the daily log file of the deployed system as the execution profile for the attacks in our deployed hosts, and use the normal events in the logs (except the events whose observed time are within the attack period) for the attacks in the DARPA TC dataset. Based on the ground truth of each attack, we manually assign lower reputation scores for the malicious files and IP addresses as required by NoDoze. Once NoDoze finishes computing the

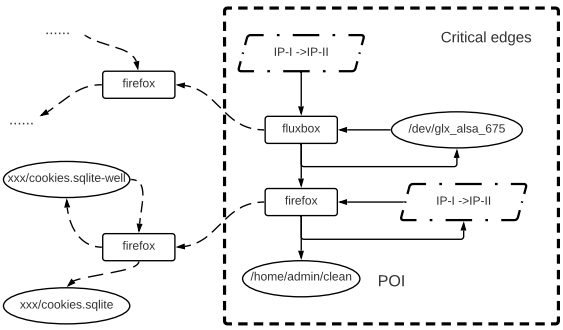

Figure 3: Critical component generated by DEPIMPACT for the "Theia Case 1" attack

anomaly scores for the whole graph, we perform the graph reduction based on the anomaly score of each path in the dependency graph.

Table 5 shows the dependency graph reduction of DEPIMPACT and the other techniques. The results show that DEPIMPACT achieves the best performance for dependency graph reduction. On average, the size of the dependency graph generated by DEPIMPACT (i.e., the critical component output by DEPIMPACT) is *at least* $106\times$ *smaller than the second-best result* (i.e., NoDoze) and three or four orders of magnitudes smaller than the other 3 techniques. We next explain the comparison with each technique.

CPR merges only the edges between pairs of nodes, and thus lack the capabilities to prune irrelevant edges originated from irrelevant system activities. Removing read-only files is heuristics-based and cannot robustly achieve good performance for different attacks as illustrated by the results (e.g., 58 for the "Wget executable" attack v.s. $600,000+$ for the "Hide File" attack). The comparison with PrioTracker shows the superiority of our *discriminative feature projection scheme* over the fanout feature in PrioTracker. From the results, we can observe that NoDoze performs generally well but poorly for certain attacks (e.g., producing graphs with $> 10,000$ edges for 5 attacks) The major reason is that there are many rare benign events in these dependency graphs that do not appear in the execution profiles. In other words, the effectiveness of NoDoze heavily relies on whether the execution profile can capture all the benign events, which is generally difficult since the runtime environment of most organizations are dynamic and versatile. On the other hand, compared to NoDoze, DEPIMPACT achieves better reduction results without sharing its two major limitations: (1) DEPIMPACT does not rely on third-party services to assign reputations to malicious files or IP addresses, which may introduce additional risks and complexity; (2) DEPIMPACT does not require the execution profile of the deployed system for training. These characteristics greatly reduce the difficulty of deploying DEPIMPACT in a new system, enabling DEPIMPACT to achieve better generalization than NoDoze.

Case Study. Figures 3 and 4 show the critical components of two attacks. We use solid lines to represent critical edges, dash lines to represent non-critical edges, and dot-and-dash lines to label attack entries. POI events are clearly marked with text descriptions.

Figure 3 shows the critical component generated by DEPIMPACT for the "Theia Case 1" attack in the DARPA TC dataset. We can observe that the firefox browser is started to download the file `/home/admin/clean` from a malicious IP address. Here, the IP addresses (i.e., the source of the backdoor) are correctly identified as attack entries, and all the critical edges are preserved.

Figure 4 shows the 3 critical components generated by DEPIMPACT for the "Shellshock" attack. The critical component of POI I (step ①) shows that Host 1 first downloads a malicious script from the C2 server, and then sends a malicious script `/tmp/crack_password.sh` to Host 2 through the process `scp`. Then, this malware collects user sensitive data in Host 2 and sends this data back to Host 1 through the process `scp` (step ②). After this step, the sensitive data is compressed in Host 1 and sent back to the C2 server (③). For this graph, the union of the 3 critical components in 2 hosts covers all the critical edges. In particular, the two special network connection edges in steps ① and ② enable the cross-host dependency tracking for revealing attack sequences.

5.3 RQ2: Selection of Entry Nodes

Intuitively, the more entry nodes DEPIMPACT uses to perform forward causality analysis, the less likely DEPIMPACT will incorrectly filter out critical edges. But using more entry nodes is likely to include more non-critical edges in the final output graph. To demonstrate the effectiveness of selecting the top-ranked entry nodes in revealing attack sequences, we show how the increase of the selected top-ranked entry nodes impact the effectiveness of DEPIMPACT in preserving critical edges and filtering out non-critical edges. As there are 3 types of system entities (i.e., processes, files, and network connections) and hundreds or even thousands of entry nodes, DEPIMPACT chooses the top-ranked entry nodes in each system-entity category and perform forward causality analysis from the nodes in the order of decreasing dependency impacts.

Evaluation Metrics. To measure the missing of critical edges, we compute the *missing rate* $M_{miss} = N_{miss}/N_{critical}$, where N_{miss} represents the number of missing critical edges and $N_{critical}$ represents the total number of critical edges (Column "Critical Edge" in Table 4). To measure the impacts of the top-ranked entry nodes, we show the edge number of the critical component DEPIMPACT in Figure 6.

Impacts on M_{miss} and Edge Number. Figure 5 and Figure 6 show the results of M_{miss} and the edge number in the critical components. As expected, when more entry nodes are used, M_{miss} decreases while the edge number increases. To reduce M_{miss} to zero, $\sim 50.0\%$ of the cases (8 attacks) use only two top-ranked nodes, and 73.3% of the cases (11 attacks) use 4 top-ranked nodes. We can also see that 6 entry nodes can cover all the attack entries for all the attack cases, and more entry nodes merely contribute to the increase of the irrelevant edges. As shown in Figure 6, there is a sharp increase of the

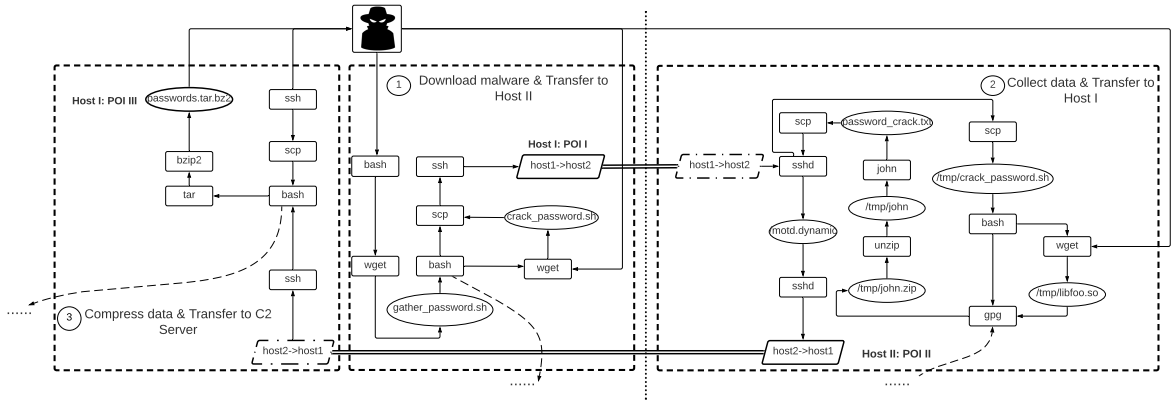

Figure 4: Critical components generated by DEPIMPACT for the "Shellshock" attack (non-critical edges are omitted). DEPIMPACT generates critical components for the three POI events and takes the union of the generated critical components, which covers all the attack steps as described in Section 5.1.2.

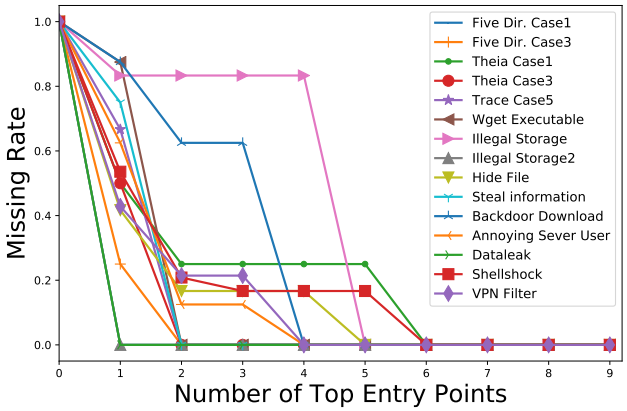

Figure 5: Missing rate using different number of top-ranked entry nodes

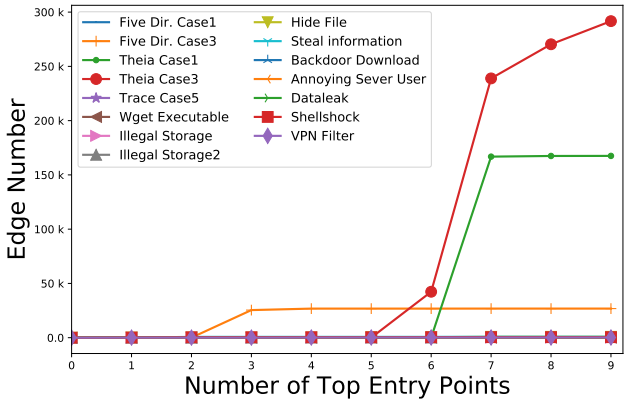

Figure 6: Edge number using different number of top-ranked entry nodes

edge numbers when more than 6 entry nodes are used. As such, when using DEPIMPACT to investigate an attack, we can first select 2 top-ranked entry nodes, and include more entry nodes one by one until adding one more entry node will result in a sharp increase of the graph size. The resulting graph typically will not miss any critical edge and still has a reasonably small size.

5.4 RQ3: Revealing Attack Entries

The evaluation results in Section 5.3 show that ranking attack entries at the top is critical for DEPIMPACT to achieve good performance (i.e., using fewer entry nodes to cover all attack entries). In this RQ, we aim to measure the effectiveness of DEPIMPACT in revealing attack entries (i.e., whether the attack entries are among the top-ranked entry nodes). In particular, we compare DEPIMPACT with 4 baseline approaches: the uniform random approach, 2 simplified versions of DEPIMPACT: DEPIMPACT-, DEPIMPACT--, and the average-projection approach. The uniform random approach ranks all the entry nodes randomly. DEPIMPACT- uses the temporal relevance and the data flow relevance to compute the dependency weight, but not the concentration ratio. DEPIMPACT-- uses only the temporal relevance to compute the dependency weight. The difference between DEPIMPACT and the average-projection approach is the parameters used to form the projection vector. For the average-projection approach, we select a fixed parameter vector $(0.334, 0.333, 0.333)$ to compute the dependency weight.

Table 7 shows the average ranks of all the attack entries computed by DEPIMPACT and the baseline approaches. We observe that DEPIMPACT consistently ranks the attack entries at the top (average rank 2.41) and achieves the best performance. Compared with DEPIMPACT--, DEPIMPACT-, the average-projection approach (shown in Column "Avg. Proj."), and the uniform random approach (shown in Column "Rand."), DEPIMPACT achieves 79.14%, 70.06%, 69.62% and 99.98% improvement in ranking the attack entries. These results demonstrate the necessity for DEPIMPACT to include all three features, and the comparison with the average-projection approach demonstrates the superiority of our discriminative feature projection scheme over a fixed parameter vector.

5.5 RQ4: System Performance

To understand the performance of DEPIMPACT, we measure the execution time of each step in DEPIMPACT, as shown

Table 6: Runtime performance of DEPIMPACT and baseline approach

Attack	Causality Ana.(s)	Edge Merge(s)	Dependency Weight Computation (s)		Dependency Impact Propagation (s)		NoDoze(s)
			DEPIMPACT	Avg. Proj.	DEPIMPACT	Avg. Proj.	
Wget Executable	120.97	0.05	0.26	0.02	0.06	0.06	1.55
Illegal Storage	92.86	0.38	7.43	0.39	19.48	47.77	173.65
Illegal Storage2	95.13	3.02	52.68	33.79	160.08	1,038.55	329.31
Hide File	223.63	42.16	463.68	16.14	1,150.35	8,486.32	899.29
Steal Information	129.82	39.51	479.02	15.98	1,157.45	8,128.28	620.87
Backdoor Download	19.74	0.44	13.87	0.32	12.75	24.05	0.71
Annoying Server User_user	17.23	0.01	0.18	0.01	0.03	0.03	0.44
Shellshcok	0.05	0.03	0.07	0.01	0.02	0.06	0.08
Dataleak	0.09	0.01	0.28	0.02	0.14	0.16	0.01
VPN Filter	0.28	0.04	0.35	0.03	0.11	0.14	0.07
Five Dir. Case1	0.81	0.01	0.10	0.01	0.04	0.02	0.02
Five Dir. Case3	2.38	0.29	9.68	0.12	1.93	2.21	39.50
Theia Case1	73.28	8.75	276.80	1.88	289.77	191.85	30.36
Theia Case3	106.17	8.34	498.81	3.06	561.95	391.96	65.88
Trace Case5	1.92	0.01	0.14	0.01	0.11	0.01	0.54
AVG	58.96	6.87	120.22	4.78	223.62	1,220.77	144.15

Table 7: Average rank of attack entries

Attack	Temp. Relv.	Temp & Data Size	Fiexd Proj.	Uni. Random	DEPIMPACT
Wget Executable	5.50	12.25	20	23.45	2
Illegal Storage	25	13	18	475.99	5
Illegal Storage2	1	1	1	1,893.66	2.50
Hide File	22	10.50	13.50	17,284.72	4
Steal Information	11	3.50	7	17,304.32	2
Backdoor Download	3.50	3.50	7.50	76.57	2
Annoying Server User	5	5	13	15.82	2
Shellshcok	11	19	13	22.63	2.30
Dataleak	35	9	9	48.34	4.30
VPN Filter	46	34	8	236.77	2.50
Five Dir. Case1	5	5	5	115.50	2
Five Dir. Case3	1	1	1	327.10	2
Theia Case1	1.5	1.5	1.5	88,956.70	1
Theia Case3	1	2	1	70,610.50	1.50
Trace Case5	2	2	2	10.10	1
AVG	11.67	8.12	8	13,160.14	2.41

in Table 6. On average, DEPIMPACT takes 343.84*s* to finish analyzing an attack (i.e., weight computation and impact propagation), and dependency graph construction (i.e., Causality Analysis) requires 58.96*s* and edge merge requires 6.87*s*. We compare DEPIMPACT with the average-projection approach for dependency weight computation and dependency impact propagation, since they share the same steps for causality analysis and edge merge. From Table 6, we observe that (1) DEPIMPACT takes more time for dependency weight computation ($\sim 120s$) because DEPIMPACT uses the Multi-KMeans++ clustering and LDA to find the optimal projection vector; (2) DEPIMPACT takes less time for dependency impact propagation. The reason is because the dependency weights computed by DEPIMPACT are much more discriminative, and hence the score propagation can converge faster. As a result, DEPIMPACT reduces the execution time by 71.94% when compared with the average-projection approach.

We also compare the execution time of DEPIMPACT (dependency weight computation plus dependency impact propagation) with the execution time of NoDoze (anomaly score computation), since they share the same causality analysis and edge merge steps. We only compare the different parts: weight computation and impact propagation. From Table 6, we can see DEPIMPACT need 343.84*s* to finish the weight computation and impact propagation, NoDoze need 144.15*s* to finish the s anomaly score computation. In particular, while DEPIMPACT requires more time for processing the 2 attacks whose dependency graphs have more than 3 million edges (i.e., the "Hide File" attack and the "Steal information" attack), DEPIMPACT produces much smaller graphs ($\sim$ 800 edges) than NoDoze ($> 20,000$ edges). On average, DEPIMPACT needs 343.84s to finish the dependency weight computation and the dependency impact propagation, and NoDoze needs 144.15s to finish the anomaly score computation (409.67*s* v.s. 209.98*s* for the whole analysis). Thus, DEPIMPACT and NoDoze have similar runtime performance for most of the attacks, and NoDoze is more efficient for certain attacks but achieves much lower graph reduction.

6 Discussion

Evasion Attacks. Existing causality analysis techniques, such as NoDoze [30], leverage execution profiles and reputations of entities (e.g., IP and file reputations) to identify anomaly edges. As shown in Section 5.2, attackers may hide their attack steps in benign events and thus their steps will be eliminated as other benign events, or try to abuse the reputation system to conceal their attack steps. Unlike existing techniques, DEPIMPACT will not suffer from this type of attacks since DEPIMPACT does not rely on execution profiles and reputations of system entities. To abuse our weight computation and back-propagation techniques, attackers may perform multiple writes to inject the complete payload into a file, with most of the writes behaving like normal behaviors. To mitigate such attacks, we may treat each of the write event as a POI event, apply DEPIMPACT on all the write events to the malicious file that contains the payload, and investigates all the generated graphs. We may also adopt process-based anomaly detection techniques [63, 70] to help distinguish these malicious writes.

Design Alternatives. DEPIMPACT is a general framework that can use different combinations of features to investigate different types of attacks. Our evaluations on a wide range of attack scenarios (Section 5.1) demonstrate the effectiveness and robustness of the chosen features. Besides the proposed features, DEPIMPACT supports easy incorporation of other

features according to specific forensic investigation needs. For edge weight computation, one alternative is to train a binary classifier using the features and output a probability score as the edge weight. However, such supervised learning-based approach faces significant limitations in our problem context: (1) as some of our features are computed with respect to the specific POI, the classification model learned for one type of attack can hardly generalize to other types of attacks with different POIs; (2) such approach typically requires large amount of training data, while our problem context is highly imbalanced in which critical edges are limited. Among unsupervised learning-based approaches, approaches based on anomaly detection [16] could be a substitution for KMeans clustering, and there could be alternatives for LDA to achieve discriminative dimensionality reduction [52, 65]. We plan to explore these options in future work.

Runtime Performance Improvement. The performance of DEPIMPACT may benefit from database optimization and parallelization. Causality analyses can be improved by adopting the database optimization techniques to speed up the search [24, 25], and can be parallelized by searching the dependency separately. Feature extraction for different edges is independent and can also be parallelized. Back-propagation (Equation (7)) can be converted into a matrix-vector product form to save CPU cycles. Further parallelization is possible by leveraging ideas similar to parallelizing PageRank [26, 43]. We plan to explore these ideas in future work.

7 Related Work

In this section, we survey three categories of related work.

Forensic Analysis via System Audit Logs. Significant progress has been made to leverage system-level audit logging for forensic analysis. Causality analysis based on system auditing data plays a critical role. King et al. [41,42] proposed a backward causality analysis technique to perform intrusion analysis by automatically reconstructing a series of events that are dependent on a user-specified POI event. Goel et al. [27] proposed a technique that recovers from an intrusion based on forensic analysis. Recent efforts have been made to mitigate the dependency explosion problem by performing fine-grained causality analysis [34, 35, 38, 46, 50], prioritizing dependencies [30, 48], customized kernel [12], and optimizing storage [33, 47, 67, 71]. However, these techniques suffer from adoption limitations as they mainly rely on heuristic rules that cause loss of information [41], intrusive system changes such as binary instrumentation [38, 50] and kernel customization [12], or execution profiles that have limited generalizations [30]. DEPIMPACT proposes to compute discriminative dependency weights based on multiple features and perform back-propagation from the POI event to compute dependency impacts for identifying attack entries, which do not share the same adoption limitations with the existing techniques. Our evaluation results further demonstrate the effectiveness of DEPIMPACT over the existing techniques.

Behavior querying leverages domain-specific languages (DSLs) to search for patterns of system call events. Gao et al. [24,25] proposed a domain-specific languages that enables efficient attack investigation by querying the historical and real-time stream of system call events. A major limitation of these DSLs is that they require manual efforts to construct the queries, which is labor-intensive and error-prone. Milajerdi et al. [54] propose to rely on the correlation of suspicious information flows to detect ongoing attack campaigns. They further propose to leverage the knowledge from cyber threat intelligence (CTI) reports to align attack behaviors recored in system auditing data [53]. Pasquier et al. [59] propose a runtime analysis of provenance by combining runtime kernel-layer reference monitor with a query module. Hossain et al. [32] propose a tag-based technique to perform real-time attack detection and reconstruction fro system auditing data. While the focuses are different, DEPIMPACT can be interoperated with these techniques to achieve a better defense.

Score Propagation. Our relevance score propagation scheme was inspired by the TrustRank algorithm [29], which was originally designed to separate spam and reputable web pages: it first selects a small set of reputable seed pages, then propagates the trust scores following the link structures using the PageRank algorithm [58], and identifies spam pages as those with low scores. Similar ideas have been applied in security and privacy application scenarios including Sybil detection [14,23,28], fake review detection [61], and attribute inference attacks [36]. DEPIMPACT is the first work that applies the score propagation idea in system audit logging domain that propagates dependency impacts to identify attack entries for filtering irrelevant dependencies.

Edge Weight Computation. Several components of DEPIMPACT are built up on a set of existing techniques. Our edge clustering step is based on Multi-KMeans++ [11], which optimizes the seed initialization for better clustering quality, compared with the standard KMeans. Our discriminative feature projection step is based on Linear Discriminant Analysis (LDA) [52], which finds a linear combination of features that characterizes or separates multiple classes of objects.

8 Conclusion

We propose DEPIMPACT, a framework that identifies the critical component of a dependency graph of a POI event generated by causality analysis, and filters out less-important dependencies introduced by irrelevant system activities. Specifically, DEPIMPACT assigns discriminative dependency weights to edges for revealing critical edges, and computes and propagates dependency impacts to entry nodes for revealing attack entries. By further performing forward causality analysis from the top-ranked entry nodes and taking the graph overlap, DEPIMPACT preserves only dependencies that are highly relevant to the POI event and attack entries. Our evaluations on real attacks demonstrate the effectiveness of DEPIMPACT in filtering out irrelevant dependencies (producing $\sim 6,250\times$ smaller graphs) while preserving the attack-relevant dependencies.

References

[1] CVE-2017-6334: WEB Netgear NETGEAR DGN2200 dnslookup.cgi Remote Command Injection. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-6334.

[2] CVE-2018-7445: NETBIOS MikroTik RouterOS SMB Buffer Overflow. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-7445.

[3] cyberkillchain. `https://www.lockheedmartin.com/en-us/capabilities/cyber/cyber-kill-chain.html`.

[4] The Equifax data breach. https://www.ftc.gov/equifax-data-breach.

[5] The Marriott data breach. https://www.consumer.ftc.gov/blog/2018/12/marriott-data-breach.

[6] Home Depot Confirms Data Breach At U.S., Canadian Stores, 2014. http://www.npr.org/2014/09/09/347007380/home-depot-confirms-data-breach-at-u-s-canadian-stores.

[7] OPM government data breach impacted 21.5 million, 2015. http://www.cnn.com/2015/07/09/politics/office-of-personnel-management-data-breach-20-million.

[8] Yahoo discloses hack of 1 billion accounts, 2016. https://techcrunch.com/2016/12/14/yahoo-discloses-hack-of-1-billion-accounts/.

[9] Schneier on Security: Router Vulnerability and the VPNFilter Botnet. `https://www.schneier.com/blog/archives/2018/06/router_vulnerab.html`, 2018.

[10] VPNFilter: New Router Malware with Destructive Capabilities. `https://symc.ly/2IPGGVE`, 2018.

[11] David Arthur and Sergei Vassilvitskii. K-means++: The advantages of careful seeding. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '07, 2007.

[12] Adam M. Bates, Dave Tian, Kevin R. B. Butler, and Thomas Moyer. Trustworthy whole-system provenance for the linux kernel. In *USENIX Security*, 2015.

[13] Matt Bishop. *Introduction to Computer Security*. Addison-Wesley Professional, 2004.

[14] Qiang C., Michael S., Xiaowei Y., and Tiago P. Aiding the detection of fake accounts in large scale social online services. USENIX, 2012.

[15] Bryan Cantrill, Adam Leventhal, and Brendan Gregg. DTrace, 2017. http://dtrace.org/.

[16] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3):15:1–15:58, July 2009.

[17] Scott A. Crosby and Dan S. Wallach. Efficient data structures for tamper-evident logging. In *USENIX Security Symposium*, page 317–334, 2009.

[18] DARPA. Transparent Computing, Defense Advanced Research Projects Agency, 2014. http://www.darpa.mil/program/transparent-computing.

[19] Exploit Database. Exploit Database, 2017. https://www.exploit-db.com/.

[20] Ebay. Ebay Inc. to ask Ebay users to change passwords, 2014. http://blog.ebay.com/ebay-inc-ask-ebay-users-change-passwords/.

[21] FireEye Inc. HammerToss: Stealthy Tactics Define a Russian Cyber Threat Group. Technical report, FireEye Inc., 2015.

[22] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics New York, NY, USA:, 2001.

[23] Peng Gao, Binghui Wang, Neil Zhenqiang Gong, Sanjeev R. Kulkarni, Kurt Thomas, and Prateek Mittal. Sybilfuse: Combining local attributes with global structure to perform robust sybil detection. In *CNS*, 2018.

[24] Peng Gao, Xusheng Xiao, Ding Li, Zhichun Li, Kangkook Jee, Zhenyu Wu, Chung Hwan Kim, Sanjeev R. Kulkarni, and Prateek Mittal. SAQL: A stream-based query system for real-time abnormal system behavior detection. In *USENIX Security*, 2018.

[25] Peng Gao, Xusheng Xiao, Zhichun Li, Fengyuan Xu, Sanjeev R. Kulkarni, and Prateek Mittal. AIQL: Enabling efficient attack investigation from system monitoring data. In *USENIX ATC*, 2018.

[26] David Gleich, Leonid Zhukov, and Pavel Berkhin. Fast parallel pagerank: A linear system approach. *Yahoo! Research Technical Report YRL-2004-038, available via http://research. yahoo. com/publication/YRL-2004-038. pdf*, 13:22, 2004.

[27] Ashvin Goel, Kenneth Po, Kamran Farhadi, Zheng Li, and Eyal de Lara. The taser intrusion recovery system. In *SOSP*, 2005.

[28] Neil Zhenqiang Gong and Di Wang. On the security of trustee-based social authentications. *Trans. Info. For. Sec.*, 9(8), 2014.

[29] Zoltán Gyöngyi, Hector Garcia-Molina, and Jan Pedersen. Combating Web Spam with Trustrank. In *Proceedings of the International Conference on Very Large Data Bases*, 2004.

[30] Wajih Ul Hassan, Shengjian Guo, Ding Li, Zhengzhang Chen, Kangkook Jee, Zhichun Li, and Adam Bates. Nodoze: Combatting threat alert fatigue with automated provenance triage. 2019.

[31] Wajih Ul Hassan, Mark Lemay, Nuraini Aguse, Adam Bates, and Thomas Moyer. Towards scalable cluster auditing through grammatical inference over provenance graphs. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, 2018.

[32] Md Nahid Hossain, Sadegh M. Milajerdi, Junao Wang, Birhanu Eshete, Rigel Gjomemo, R. Sekar, Scott D. Stoller, and V. N. Venkatakrishnan. SLEUTH: real-time attack scenario reconstruction from COTS audit data. In *USENIX Security Symposium*, pages 487–504, 2017.

[33] Md Nahid Hossain, Junao Wang, R. Sekar, and Scott D. Stoller. Dependence-preserving data compaction for scalable forensic analysis. In *USENIX Security Symposium*, pages 1723–1740, 2018.

[34] Yang Ji, Sangho Lee, Evan Downing, Weiren Wang, Mattia Fazzini, Taesoo Kim, Alessandro Orso, and Wenke Lee. Rain: Refinable attack investigation with on-demand inter-process information flow tracking. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 377–390. ACM, 2017.

[35] Yang Ji, Sangho Lee, Mattia Fazzini, Joey Allen, Evan Downing, Taesoo Kim, Alessandro Orso, and Wenke Lee. Enabling refinable cross-host attack investigation with efficient data flow tagging and tracking. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1705–1722, 2018.

[36] Jinyuan Jia, Binghui Wang, Le Zhang, and Neil Zhenqiang Gong. Attriinfer: Inferring user attributes in online social networks using markov random fields. In *WWW*, 2017.

[37] Xuxian Jiang, AAron Walters, Dongyan Xu, Eugene H. Spafford, Florian Buchholz, and Yi-Min Wang. Provenance-aware tracing of worm break-in and contaminations: A process coloring approach. In *ICDCS*, 2006.

[38] Yonghwi K., Fei W., Weihang W., Kyu H. L., Wen-C. L., Shiqing M., Xiangyu Z., Dongyan X., Somesh J., Gabriela F. C., Ashish G., and Vinod Y. MCI : Modeling-based causality inference in audit logging for attack investigation. In *NDSS*, 2018.

[39] Amin Kharraz, Sajjad Arshad, Collin Mulliner, William K. Robertson, and Engin Kirda. UNVEIL: A large-scale, automated approach to detecting ransomware. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 757–772, 2016.

[40] Taesoo Kim, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Intrusion recovery using selective re-execution. In *OSDI*, 2010.

[41] Samuel T. King and Peter M. Chen. Backtracking intrusions. In *SOSP*, 2003.

[42] Samuel T. King, Zhuoqing Morley Mao, Dominic G. Lucchetti, and Peter M. Chen. Enriching intrusion alerts through multi-host causality. In *NDSS*, 2005.

[43] Christian Kohlschütter, Paul-Alexandru Chirita, and Wolfgang Nejdl. Efficient parallel computation of pagerank. In *European Conference on Information Retrieval*, pages 241–252. Springer, 2006.

[44] Christopher Kruegel, Fredrik Valeur, and Giovanni Vigna. *Intrusion Detection and Correlation - Challenges and Solutions*, volume 14 of *Advances in Information Security*. Springer, 2005.

[45] Yonghwi Kwon, Fei Wang, Weihang Wang, Kyu Hyung Lee, Wen-Chuan Lee, Shiqing Ma, Xiangyu Zhang, Dongyan Xu, Somesh Jha, Gabriela F Ciocarlie, et al. Mci: Modeling-based causality inference in audit logging for attack investigation. In *NDSS*, 2018.

[46] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. High accuracy attack provenance via binary-based execution partition. In *NDSS*, 2013.

[47] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. Loggc: garbage collecting audit log. In *CCS*, 2013.

[48] Yushan Liu, Mu Zhang, Ding Li, Kangkook Jee, Zhichun Li, Zhenyu Wu, Junghwan Rhee, and Prateek Mittal. Towards a timely causality analysis for enterprise security. In *NDSS*, 2018.

[49] Shiqing Ma, Juan Zhai, Yonghwi Kwon, Kyu Hyung Lee, Xiangyu Zhang, Gabriela F. Ciocarlie, Ashish Gehani, Vinod Yegneswaran, Dongyan Xu, and Somesh Jha. Kernel-supported cost-effective audit logging for causality tracking. In *USENIX ATC*, 2018.

[50] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. Protracer: towards practical provenance tracing by alternating between logging and tainting. 2016.

[51] Microsoft. ETW events in the common language runtime, 2017. https://msdn.microsoft.com/en-us/library/ff357719(v=vs.110).aspx.

[52] Sebastian Mika, Gunnar Ratsch, Jason Weston, Bernhard Scholkopf, and Klaus-Robert Muller. Fisher discriminant analysis with kernels, 1999.

[53] Sadegh M. Milajerdi, Birhanu Eshete, Rigel Gjomemo, and V.N. Venkatakrishnan. Poirot: Aligning attack behavior with kernel audit records for cyber threat hunting. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, page 1795–1812, 2019.

[54] Sadegh M Milajerdi, Rigel Gjomemo, Birhanu Eshete, R Sekar, and VN Venkatakrishnan. HOLMES: real-time APT detection through correlation of suspicious information flows. 2019.

[55] MITRE. Common Vulnerabilities and Exposures (CVE), 2020. https://cve.mitre.org/.

[56] netwrix. Insider threat detection, 2020. https://www.netwrix.com/insider threat detection.html.

[57] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[58] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical Report 1999-66, 1999.

[59] Thomas Pasquier, Xueyuan Han, Thomas Moyer, Adam Bates, Olivier Hermant, David Eyers, Jean Bacon, and Margo Seltzer. Runtime Analysis of Whole-system Provenance. In *ACM CCS*. ACM, 2018.

[60] Tadeusz Pietraszek. Using adaptive alert classification to reduce false positives in intrusion detection. In Erland Jonsson, Alfonso Valdes, and Magnus Almgren, editors, *Recent Advances in Intrusion Detection*, pages 102–124. Springer Berlin Heidelberg, 2004.

[61] Shebuti Rayana and Leman Akoglu. Collective opinion spam detection: Bridging review networks and metadata. In *KDD*, 2015.

[62] Redhat. The linux audit framework, 2017. https://github.com/linux-audit/.

[63] Suphannee Sivakorn, Kangkook Jee, Yixin Sun, Lauri Kort-Parn, Zhichun Li, Cristian Lumezanu, Zhenyu Wu, Lu-An Tang, and Ding Li. Countering malicious processes with process-dns association. In *Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2019.

[64] Georgios P. Spathoulas and Sokratis K. Katsikas. Reducing false positives in intrusion detection systems. *Computers & Security*, 29(1):35–44, 2010.

[65] Masashi Sugiyama. Local fisher discriminant analysis for supervised dimensionality reduction. In *Proceedings of the 23rd international conference on Machine learning*. ACM, 2006.

[66] Sysdig. Sysdig, 2017. https://sysdig.com/.

[67] Yutao Tang, Ding Li, Zhichun Li, Mu Zhang, Kangkook Jee, Xusheng Xiao, Zhenyu Wu, Junghwan Rhee, Fengyuan Xu, and Qun Li. Nodemerge: Template based efficient data reduction for big-data causality analysis. In *ACM CCS*, 2018.

[68] New York Times. Target data breach incident, 2014. http://www.nytimes.com/2014/02/27/business/target-reports-on-fourth-quarter-earnings.html?_r=1.

[69] David A. Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 255–264, 2002.

[70] Shen Wang, Zhengzhang Chen, Xiao Yu, Ding Li, Jingchao Ni, Lu-An Tang, Jiaping Gui, Zhichun Li, Haifeng Chen, and Philip S. Yu. Heterogeneous graph matching networks for unknown malware detection. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 3762–3770, 2019.

[71] Zhang Xu, Zhenyu Wu, Zhichun Li, Kangkook Jee, Junghwan Rhee, Xusheng Xiao, Fengyuan Xu, Haining Wang, and Guofei Jiang. High fidelity data reduction for big data security dependency analyses. In *CCS*, 2016.