DEPIMPACT: 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 for creating a suspicious file), causality analysis constructs a dependency graph, where 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 present a different set of properties (e.g., data flow and time) compared to 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 from the original dependency graph that are not found in the forward causality analysis. Our evaluations on the 100 million real system auditing events of 10 attacks show that DEPIMPACT can significantly reduce the large dependency graphs ($\sim 700,000$ edges) to a small graph (~ 260 edges), which is $\sim 2,700\times$ smaller. The comparison of DEPIMPACT with other four state-of-the-art causality analysis techniques shows that DEPIMPACT is at least $33\times$ more effective in reducing the dependency graphs for revealing the attack sequences.

I. INTRODUCTION

Recent cyber attacks have plagued many well-protected businesses, causing significant financial losses [1]–[7]. 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 [8]–[15]. System monitoring collects kernel auditing events about system calls as system audit logs. The collected data enables approaches based on *causality analysis* [8], [9], [12], [13], [16]–[19] 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 [18], [20], [21] and assisting timely system recovery [16], [17].

While the research on attack investigation based on causality analysis shows great potential, existing approaches require non-trivial efforts of inspection [18], [22], 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 (e.g., an alert event). Such contextual information is particularly effective in distinguishing benign and attack-related events such as distinguishing benign uses of *ZIP* from ransomware [18], [23]. However, due to the dependency explosion problem [24]–[26], it is hard for security analysts to understand a huge graph (typically containing >100,000 edges [18], [22]), and find the edges that are critical to the attack (i.e., finding damaging needles in a very large haystack).

Key Insight. Existing work [8], [9], [12], [13], [16]–[19] mainly relies on the event time to identify causal dependencies (e.g., writing a file before reading it) and the dependency graph contains many dependencies that are less important to attack investigation. By carefully inspecting the dependency graphs of various attacks [12]–[14], [19], 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 present a different set of properties (e.g., amount of data flow) and are more related to the POI event in these properties. Second, a POI event is often caused by a few sources, referred to as *attack entries*. These attack entries are entry points of the attack sequence that leads 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. If we manage to distinguish critical edges and attack entries from non-critical edges and irrelevant entries, the size of the dependency graph can be greatly reduced while preserving the attack sequence, which facilitates attack investigation. However, due to the dependency explosion problem, there are two major challenges for achieving such goals.

As illustrated previously, the large number of non-critical edges come from the less-important dependencies brought by causality analysis, and the less-important dependencies then come from irrelevant system activities performed by processes that are causally related to the POI event. Furthermore, 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 V-A). As a result, manually inspecting these daunting number of edges and entry nodes requires a significant amount of efforts.

The problem becomes even more challenging when different POI events are selected for investigation and different attack scenarios are considered. While existing techniques have made attempts to address these problems, they mainly rely on heuristic rules that cause loss of information [8], intrusive system changes [12], [19] such as binary instrumentation and kernel customization, or execution profiles [18], hindering their practical adoption.

To facilitate attack investigation, there is a pressing need for automated techniques to reveal critical edges and attack entries. The challenge is to design a general solution framework that (1) encapsulates such automated techniques, (2) is broadly applicable to different POI events and diversified attack scenarios, and (3) does not suffer from the same adoption limitations as existing techniques.

Contributions. Based on the key insights, we propose DEPIMPACT, a framework that *facilitates attack investigation by revealing critical edges and attack entries, 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 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 (Section V-B), which makes it easy for security analysts to obtain the contextual information of the attack. For example, security analysts can inspect attack entries to identify the entry points of the attack, and inspect critical edges to gain visibility into the attack sequence.

In particular, to address the aforementioned challenges, DEPIMPACT employs three major techniques as follows:

(1) Dependency Weight Computation: To capture the differences between critical edges and non-critical edges, for each edge, DEPIMPACT extracts multiple features (e.g., time, data flow amount, node degree) that capture such differences from multiple aspects (Section IV-B2). Then, DEPIMPACT employs a *discriminative feature projection scheme* based on Linear Discriminant Analysis (LDA) [27] to compute a weight score from the features, referred to as *dependency weight* (Section IV-B3). Dependency weight, by definition, is a score in the range $[0.0, 1.0]$ that models the coupling strength between the dependency 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. Our feature projection scheme finds an optimal projection plane for the features, so that the projected dependency weights of critical edges and non-critical edges are maximally separated. Using dependency weight, critical edges and non-critical edges can be distinguished, and the dependency graph becomes a weighted dependency graph that encodes such differences.

(2) Dependency Impact Back-Propagation & Entry Node Ranking: To reveal attack entries, DEPIMPACT employs a notion of *dependency impact*. Dependency impact, by definition, is a score[1] for each node in the dependency graph 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 (1.0 by default for both source node and sink node) in the POI event backward along the edges to all entry nodes. Inspired by TrustRank [28], our score propagation scheme computes the dependency impact of a node using a weighted sum of its child nodes' dependency impacts, and the weights are the normalized dependency weights of the node's outgoing edges. The intuition behind our score propagation scheme is that for an attack entry that has a high impact on the POI event, its child nodes on the attack sequence should also have a high impact, proportionally by the dependency weight of the edges that connect to the child nodes. By taking the normalization, we guarantee that the node's impact is not greater than the maximum of its child nodes' impacts, so that the dependency impact of a node will not be over 1.0 and will gradually degrade through the propagation. DEPIMPACT iteratively updates the scores for all the nodes until a stable point is reached. After propagation, DEPIMPACT ranks the entry nodes based on their dependency impacts, and the top-ranked entry nodes are 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 overlap between the forward graph and the original backward dependency graph produced by the backward causality analysis from the POI event well preserves the nodes and edges that are highly related to both the POI event and the attack entries. Such overlap is referred to as the *critical component* of the original dependency graph, which preserves the critical attack information at a reduced size.

Evaluation. We implemented DEPIMPACT (∼20K lines of code) and deployed it on a server to collect real system monitoring data. We performed 7 attacks that are used in prior studies [13], [25], [29], [30] and 3 multi-step intrusive attacks based on the Cyber Kill Chain framework [31] and CVE [32], and applied DEPIMPACT to investigate them. During our evaluation, the deployed hosts continue to resume their routine

[1]The value of dependency impact is in the range $[0.0, 1.0]$, with 1.0 representing the max impact.

tasks to emulate the real-world deployment where irrelevant system activities and attack activities co-exist. In total, we collected ∼100 million system auditing events.

The evaluation results show that DEPIMPACT is highly effective in revealing critical edges and attack entries. On average, the size of the critical component produced by DEPIMPACT has ~ 260 edges, which is $\sim 2,700\times$ smaller than the size of the original dependency graph ($\sim 700,000$ edges) produced by directly applying causality analysis. 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 (*3.13 *on average)*, demonstrating the effectiveness in using the top-ranked entry nodes for identifying critical components. Compared with the average-projection approach that uses an average projection vector for computing dependency impacts, DEPIMPACT achieves 29.34% improvement in ranking attack entries, demonstrating the superiority of DEPIMPACT's discriminative feature projection scheme.

The comparison with four other state-of-the-art causality analysis techniques (CPR [25], ReadOnly [33], PrioTracker [13], and NoDoze [18]) shows that DEPIMPACT is *at least* $33\times$ *more effective in dependency graph reduction*. Furthermore, DEPIMPACT does not share the same adoption limitations as these techniques (e.g., training on an execution profile and reputation assignment [18]).

Finally, DEPIMPACT finishes analyzing an attack within 8 minutes, which is $\sim 5\times$ faster when compared with the average-projection approach. While DEPIMPACT needs more time for dependency weight computation using clustering and LDA ($\sim 102s$), the average-projection approach fails to distinguish critical edges and non-critical edges due to poor weight assignment, and spends a significantly more time in score propagation ($\sim 1700s$; the propagation takes more iterations to converge). DEPIMPACT and the state-of-the-art technique, NoDoze, have similar runtime performance for most of the attacks. Though on average DEPIMPACT is slightly slower than NoDoze (~ 5 minutes), DEPIMPACT produces much smaller graphs for the attacks that DEPIMPACT takes longer time to analyze (~ 800 edges v.s. $> 20,000$ edges), which facilitates further attack investigation.

II. BACKGROUND AND MOTIVATION

A. 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 [8]–[16], [18], 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*.

TABLE I: 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 II: 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

Both entities and events have critical security-related attributes (Tables I and II). 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).

B. Causality Analysis

Causality analysis [8], [9], [12], [13], [16]–[19] 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.

C. 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

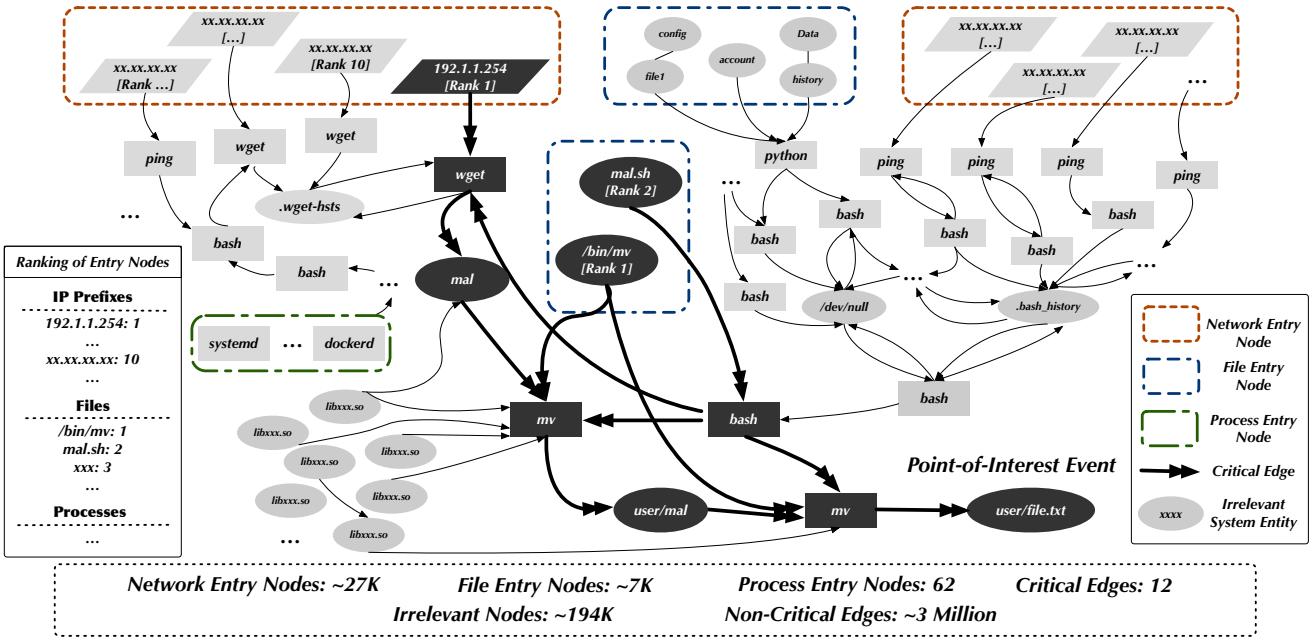

Fig. 1: Partial dependency graph of an attack that downloads a malicious file and hides the file by renaming it (rectangles for processes, ovals for files, parallelograms for network connections). The complete dependency graph constructed from the POI event (renaming to `user/file.txt`) via backward 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.

of critical edges (i.e., 12) are buried in an overwhelmingly large number ($\sim$ 3 million) of non-critical edges (i.e., less-important dependencies), and same for attack entries (i.e., 2 out of $\sim 35K$ irrelevant entry nodes).

Using DEPIMPACT to Identify Critical Component. To remove less-important dependencies introduced by irrelevant system activities, DEPIMPACT identifies the critical component, a subgraph that consists of mainly critical edges and attack entries, by (1) computing dependency weights for edges and computing dependency impacts for entry nodes, (2) ranking entry nodes based on their dependency impacts, and (3) performing forward causality analysis from the top-ranked entry nodes to filter out irrelevant parts of the graph.

In this example, we divide the entry nodes into 3 categories (i.e., network connections, files, and processes), and rank 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 of 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.

III. OVERVIEW

Figure 2 shows the architecture of DEPIMPACT. Given a POI event, DEPIMPACT employs automated techniques to identify the critical component of the dependency graph for the POI event 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 [34]–[37] to collect system-level audit logs about system calls (Section IV-A1). Given a POI event, DEPIMPACT parses the collected logs and performs backward causality analysis [8], [9] to generate a backward dependency graph for the POI event. (Section IV-A2).

In Phase II, DEPIMPACT first employs state-of-the-art dependency graph reduction techniques [25] to reduce the graph size (Section IV-B1). Then, DEPIMPACT extracts features for edges (Section IV-B2) and employs a discriminative feature projection scheme based on LDA to compute dependency weights (Section IV-B3) 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 (Section IV-C1). Then, DEPIMPACT ranks entry nodes based on their dependency impacts and selects the top candidates (Section IV-C2). 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 (Section IV-C3). Compared with the original backward dependency graph, the critical component well preserves the information that is critical to attack investigation (critical edges and attack entries) at a reduced size.

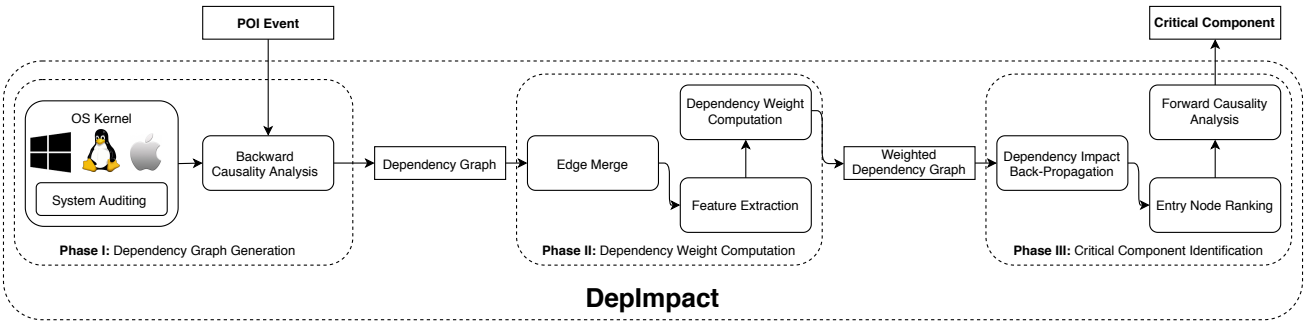

Fig. 2: Architecture of DEPIMPACT

TABLE III: 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

Threat Model. Our threat model is similar to the threat model of previous work on system monitoring [8], [9], [13]–[15], [18], [33]. We assume that kernel and kernel-layer auditing framework [34]–[37] are part of our trusted computing base (TCB), and existing software and kernel hardening techniques [38], [39] 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) that do not go through kernel-layer auditing and thus cannot be captured by the underlying provenance tracker, We also do not consider Mimicry attacks [40] where attackers deliberately evade intrusion detection systems through a chain of events that seem benign in enterprises. Existing intrusion detection systems [41]–[43] often relies on heuristics or analysis based on the properties of a single event, and thus are vulnerable for such attack. 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.

IV. DESIGN OF DEPIMPACT

In this section, we present DEPIMPACT in detail.

A. Phase I: Dependency Graph Generation

1) System Auditing: DEPIMPACT leverages mature system auditing frameworks [34]–[37] to collect system-level 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 III 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 I and II show the representative attributes of entities and events extracted by DEPIMPACT. Furthermore, 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, as processes usually communicate with some servers using different network connections but with the same IPs and ports, treating these connections differently greatly increases the amount of data we trace and such granularity is not required in most of the cases [13]–[15]. Thus, 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.

2) Backward Causality Analysis: Given a POI event, DEPIMPACT performs backward causality analysis (Section II-B) 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.

B. Phase II: Dependency Weight Computation

1) Edge Merge: The dependency graph produced by causality analysis often has many edges between the same pair of nodes [25]. The reason for generating these excessive edges is that OS typically finishes a read/write task (e.g., file read/write) by distributing the data to multiple system calls and each system call processes only a portion of data. Inspired by the recent work that proposed Causality Preserved Reduction (CPR) [25] for dependency graph reduction, DEPIMPACT merges the edges between two nodes. As shown in [25], CPR does not work well for processes that have many interleaved read and write system calls, which introduces excessive causality. As such, DEPIMPACT adopts a more aggressive approach: for edges with the same direction (i.e., representing read or write) between two nodes, DEPIMPACT will merge them into one edge 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 [25]. Since such merge is performed after the dependency graph generation, all the dependencies are still preserved with the time windows of certain edges merged.

2) Feature Extraction: For each edge, DEPIMPACT extracts three features to compute a dependency weight, which models the coupling strength (i.e., level of relevance) 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 Size Relevance $f_{S(e)}$. Intuitively, edges that have relatively the same size are more likely to be relevant. As such, we design feature $f_{D(e)}$ to model the data size relevance of an edge $e(u, v)$ to the POI event:

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

where s_e and s_{e_s} represent the data size 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 size relevance $f_{S(e)}$. Note that we use a small positive number α (we set $\alpha = 1\text{e}{-4}$) 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., $1\text{e}{-10}$) to compute its feature value: $f_{T(e_s)} = ln(1 + 1\text{e}10)$. 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.

It is important to note that 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 V-A) demonstrate the effectiveness and robustness of the chosen features. To analyze more types of attacks, new features can be incorporated seamlessly: network attacks may need features like protocol type and port number; illegal file operation may need features like the owner and last modification time.

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 [44]. Instead of naively taking the average, DEPIMPACT employs a *discriminative feature projection scheme* based on Linear Discriminant Analysis (LDA) [27] to compute an optimal projection vector. This way, the projected weights of critical edges and non-critical edges are maximally separated, and critical edges have higher weights than non-critical edges. 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 [44], and then employs Multi-KMeans++ clustering algorithm [45]. 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, in the second step, DEPIMPACT leverages Linear Discriminant Analysis (LDA) [27] 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

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 $\quad\quad\quad\quad res\ +\!= v.score * G.edge(u,v).weight$
11 $\quad\quad\quad diff\ +\!= |u.score - res|$
12 $\quad\quad\quad u.score \leftarrow res$

the number of non-critical edges (as can be seen from attack cases in Section V-A). 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 V-D, 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 IV-C), 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.

C. Phase III: Critical Component Identification

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 [46], our scheme preserves the scores along long dependency paths and prevents fast degradation.

Algorithm 1 illustrates our dependency 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 $\delta = 1e - 13$ as it gives robust results from our evaluations.

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 that we want to identify. By selecting the top-ranked candidates and performing forward causality analysis to identify descendants, we are able to significantly prune the dependency graph while preserving the critical parts.

In the current design, we have a special treatment of system library nodes. As has been shown in prior work [26], 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.

3) Forward Causality Analysis & Critical Component Identification: From the top-ranked entry nodes, DEPIMPACT performs forward causality analysis until reaching the POI event. The process is similar to the backward causality analysis as illustrated in Section IV-A2. 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.

V. Evaluation

We built DepImpact (∼20K lines of code) upon Sysdig [37], and deployed our tool on 2 hosts to collect system auditing events and perform attack investigation. 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. We performed a series of attacks based on known exploits [13], [25], [29], [30] in the deployed environment, and applied DepImpact to investigate these attacks, demonstrating the effectiveness of DepImpact. In total, our evaluations used real system audit logs that contain *100 million* events.

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 do the top-ranked entry nodes affect DepImpact in revealing attack sequences?
- **RQ3**: How effective is DepImpact in revealing attack entries?
- **RQ4**: How efficient is DepImpact in investigating an attack?

RQ1 aims to evaluate the overall effectiveness of DepImpact in dependency graph reduction, and compare DepImpact with other state-of-the-art causality analysis techniques. RQ2 aims to evaluate how the top-ranked entry nodes affect the effectiveness of DepImpact. RQ3 aims to evaluate whether DepImpact consistently ranks the attack entries upfront, and compare DepImpact with other baseline approaches. RQ4 aims to measure the execution times of DepImpact and its variation, and compare DepImpact with other state-of-the-art causality analysis techniques.

A. Evaluation Setup

To evaluate DepImpact, we performed 10 attacks in the deployed environment: 7 attacks based on commonly used exploits and 3 multi-step intrusive attacks based on the Cyber Kill Chain framework [31] and CVE [32]. We then collected system auditing events for the attacks and applied DepImpact to analyze the events. DepImpact is executed on a server with an Intel(R) Xeon(R) CPU E5-2637 v4 (3.50GHz), 256GB RAM running 64bit Ubuntu 18.04.1. Next, we describe the attacks in detail.

1) Attacks Based on Commonly Used Exploits: These 7 attacks are used as test cases in prior work [13], [25], [29], [30], 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.

2) Multi-Step Intrusive Attacks: These 3 multi-step intrusive attacks capture the important traits of attacks depicted in the Cyber Kill Chain framework [31] and CVE [32]. Note that some steps of these attacks may not be fully captured by system auditing (e.g., user inputs and inter-process communications). Such limitations can be addressed by employing more powerful auditing tools, which is orthogonal this paper.

Attack 1: Password Cracking After Shellshock Penetration. After the initial shellshock penetration, the attacker first connects to Cloud services (e.g., Dropbox, Twitter) and downloads an image where C2 (Command and Control) host's IP address is encoded in the EXIF metadata. The behavior is a common practice shared by APT attacks [47], [48] to evade the network-based detection system based on DNS blacklisting.

Using the IP, the malware connects to the C2 host. The C2 host directs the malware to take some lateral movements, including a series of stealthy reconnaissance maneuvers. In this stage, the attacker generally takes a number of actions. Among those, we emulate the password cracking attack. The attacker downloads password cracker payload and runs it against password shadow files.

Attack 2: Data Leakage After Shellshock Penetration. After lateral movements, the attacker attempts to steal the valuable assets from the host. This stage mainly involves the behaviors of local and remote file system scanning activities, copying and compressing of important files, and transferring the files to its C2 host. The attacker scans the file system, scrap files into a single compress file and transfer it back to C2 host.

Attack 3: VPN Filter. We prototyped a famous IoT attack campaign: VPN Filter malware [49], which infected millions of IoT devices by exploiting a number of known or zero-day vulnerabilities [50], [51]. The attack's significance lies in

TABLE IV: Statistics of dependency graphs generated for the 10 attacks

Attack	Causality Analysis # V	Causality Analysis # E	Edge Merge # V	Edge Merge # E	Entry Nodes	Critical Edge	Attack Entries
Wget Executable	126	673	126	363	46	8	2
Illegal Storage	8450	93085	8450	62073	960	6	2
Illegal Storage 2	42450	658913	42450	378326	3499	4	2
Hide File	194208	6464098	194208	3273769	35203	12	2
Steal Information	195636	6493626	195636	3291208	35213	4	2
Backdoor Download	7510	69479	7510	60390	157	8	2
Annoying Server User	114	585	114	318	34	8	2
Shellshock	81	10289	81	229	46	13	2
Dataleak	174	734	174	459	95	11	1
VPN Filter	549	2986	549	661	500	5	3
AVG	44929.8	1379446.8	44929.8	706779.6	7575.3	7.9	2

TABLE V: Dependency graph reduction of DEPIMPACT and baseline approaches (# Edges)

Attack	CPR	ReadOnly	PrioTracker	NoDoze	DEPIMPACT
Wget Executable	363	58	58	288	49
Illegal Storage	62073	16211	6948	10260	67
Illegal Storage 2	378326	89779	37112	19512	607
Hide File	3273769	613303	114614	37253	805
Steal Information	3291208	618025	115223	20426	854
Backdoor Download	60390	15990	6024	269	59
Annoying Server User	318	56	39	227	21
Shellshcok	229	88	101	27	49
Dataleak	459	73	31	212	24
VPN Filter	661	56	552	163	25
Avg	706779.6	135363.9	28070.2	8863.7	256

how the malware operates during its lateral movement stage following its initial penetration. The campaign employs up-to-date hacker practices to bypass conventional security solutions based on static blacklisting approaches and has an architecture to download the plug-in payload on-demand at run-time. We prototyped the malware by referring to one of its sample for x86 architecture [52].

The VPN Filter stage 1 malware accesses a public image repository to get an image. In the EXIF metadata of the image, it contains the IP address for the stage 2 host. It downloads the VPN Filter stage2 from the stage2 server, and executes it to launch the attack.

Dependency Graph Statistics for the 10 Attacks. Table IV shows the statistics of the generated dependency graphs for the 10 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 IV-B1). 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 attack entries. We clearly observe that even after edge merges, there still remains a large number of edges in the dependency graphs (707K on average with the max being 3.3 million edges), which motivates the further pruning provided by DEPIMPACT.

B. 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 [25], ReadOnly [33], PrioTracker [13], and NoDoze [18]. 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 V-C). 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. Paths having higher anomaly scores will be reported. We use the daily log file of the deployed system as the execution profile required by NoDoze and compute each path's anomaly score accordingly. Because we have 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 anomaly scores for the whole graph, we perform the graph reduction based on the anomaly score of each path in the dependency graph. Note that it is fairly easy for a technique to keep all the critical edges by keeping all the edges generated by the causality analysis from the POI event, but it is far more difficult to preserve the critical edges and filter the non-critical edges at the same time. Thus, we tune the parameters of all the techniques to preserve all the critical edges whenever possible, and compare the results of dependency graph reduction in terms of the number of edges.

Table V shows the dependency graph reduction of DEPIMPACT and other approaches. The results clearly 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* $33\times$ *smaller than the second-best result* (i.e., NoDoze) and three or four orders of magnitudes smaller than the other 3 methods. We next explain the results comparison with each technique.

DEPIMPACT is built upon CPR and thus the results demonstrate the better pruning power brought by the forward causality analysis from the top-ranked entry nodes. Removing read-only files is heuristics-based and cannot robustly achieve good performance for different attacks as illustrated by the results (e.g., 56 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 generally performs well but perform poorly for certain attacks, e.g., the "Hide File" attack and the "Steal information" attack, whose dependency graphs have more than 3 million edges.

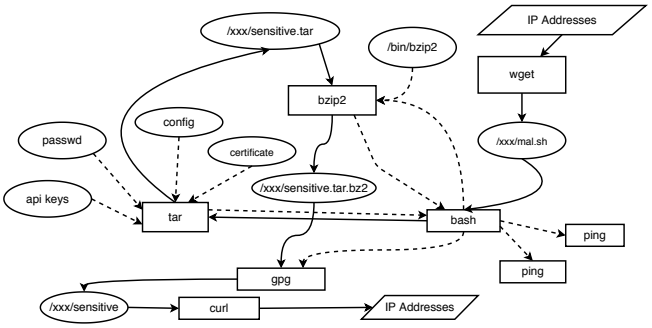

Fig. 3: Critical component (reduced dependency graph) generated by DEPIMPACT for the "Dataleak" attack

The major reason is that there are many rare benign events in these dependency graphs that do not appear in the execution profiles used for training. In NoDoze, the anomaly score of a given path is the product of each event's probability along the path. These rare benign events cause many paths that include these benign events to have higher anomaly scores, which greatly degrades the effectiveness of NoDoze. 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 quite 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 for 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. Figure 3 shows the critical component generated by DEPIMPACT for the "Dataleak" attack. We use solid lines to represent the critical edges and dash lines to represent non-critical edges. From Figure 3, we can observe that a malicious script `mal.sh` is downloaded from a suspicious external IP address. This script starts a bash and then compresses the sensitive user data using `tar`. After the compression, the attacker further compresses this file and encrypts it using `bzip2` and `gpg`. Then, the encrypted file is sent to the attacker host using `curl`. For this attack, the attack sequence consists of 11 edges and DEPIMPACT generates a graph of 24 edges that preserves all the 11 edges, while NoDoze generates 212 edges and the other techniques generates 31 edges at the best (i.e., PrioTracker), demonstrating the superiority of DEPIMPACT.

C. RQ2: Impacts of Top-Ranked Entry Nodes

Intuitively, the more entry nodes DEPIMPACT uses to perform forward causality analysis, the less likely DEPIMPACT will incorrectly filter out critical edges. However, more entry nodes will also cause more non-critical edges to be preserved 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 attack entries may be from different types of system entities, we also compare the effectiveness of two mechanisms in selecting top-ranked entry nodes: one considers the types of system entities and the other does not.

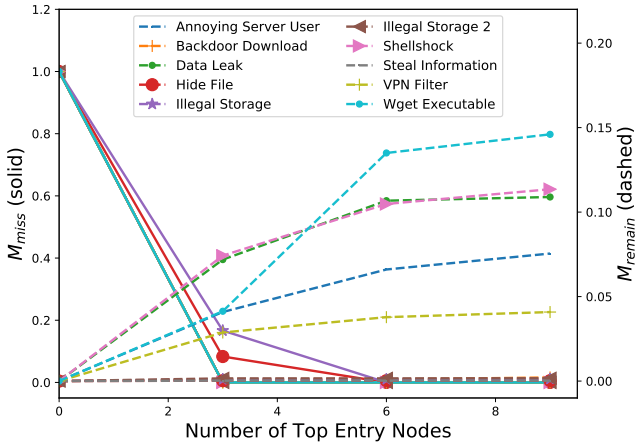

Fig. 4: M_{miss} and M_{remain} using top-ranked entry nodes by considering 3 types of system entities (3, 6, 9 nodes)

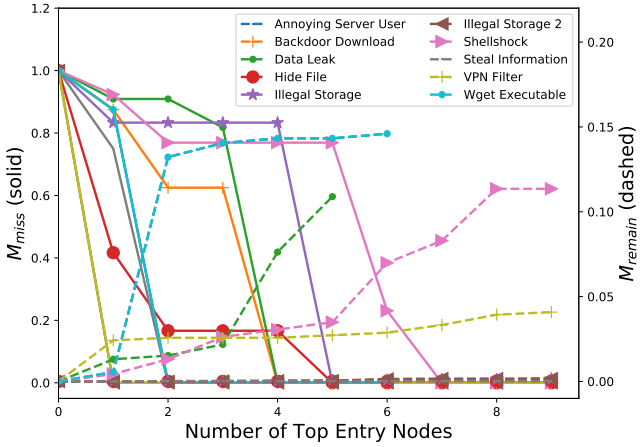

Fig. 5: M_{miss} and M_{remain} using top-ranked entry nodes without considering the types of system entities (1-9 nodes)

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 IV). To measure the effectiveness of graph reduction, we compute the *remaining rate* $M_{remain} = N_{remain}/N_{total}$, where N_{remain} represents the number of edges in the output graph and N_{total} represents the number of edges in the dependency graph after the edge merge (Column "Edge Mer. # E" in Table IV).

Impacts on M_{miss} and M_{remain}. As there are 3 types of system entities (i.e., processes, files, and network connections), DEPIMPACT uses the top-ranked entry nodes in different system entity categories to perform forward causality analysis (i.e., special treatment in Section IV-C2). Figure 4 shows the values of M_{miss} and M_{remain} with the increases of the used

TABLE VI: Average rank of attack entries

Attack	DEPIMPACT--	DEPIMPACT-	Avg. Proj.	Rand.	DEPIMPACT
Wget Executable	9.50	12.50	7.00	23.45	6.00
Illegal Storage	13.00	13.00	7.00	475.99	3.00
Illegal Storage 2	1.00	1.00	1.00	1,893.66	1.00
Hide File	15.00	10.50	3.00	17,284.72	1.50
Steal Information	6.50	3.50	3.00	17,304.32	3.00
Backdoor Download	6.50	7.00	7.50	76.57	6.50
Annoying Server User	2.50	4.50	9.50	15.82	4.00
Shellshock	1.00	1.00	1.00	22.63	1.00
Dataleak	1.00	1.00	1.00	48.34	1.00
VPN Filter	5.00	5.00	4.30	236.77	4.30
AVG	6.10	5.90	4.43	3,738.23	3.13

entry nodes. As expected, when more entry nodes are used, M_{miss} decreases while M_{remain} increases. This is because performing the forward causality analysis from more entry nodes will make the final graph overlap larger, which is likely to include more critical edges and more non-critical edges. We can clearly see that when 6 entry nodes are used (2 nodes from each of the 3 system entity categories), M_{miss} becomes 0.0. We further confirm that these 6 entry nodes cover all the attack entries, and more entry nodes merely contribute to the increase of M_{remain}. Another finding is that the dependency impact of the node ranked at the third is about 70% of the top 1 node's dependency impact. Thus, alternatively, we can use 70% of the top 1 node's dependency impact as the threshold to choose the top-ranked entry nodes for filtering.

Comparison of Two Selection Mechanisms for Top-Ranked Entry Nodes. We compare DEPIMPACT's entry node selection mechanism with another mechanism that does not consider system entity categories (i.e., selecting top-ranked entry nodes based on their dependency impacts directly). Figure 5 shows the values of M_{miss} and M_{remain} for this mechanism. As we can see, for some attacks (e.g., the "VPNFilter" and "Wget Executable" attacks), M_{miss} becomes 0.0 when the top 1 and the top 2 entry nodes are used, but some attacks (e.g., the "Shellshock" attack) requires 7 entry nodes for M_{miss} to become 0.0. On the contrary, for all the attacks, the selection mechanism that considers system types can ensure zero-loss of critical edges when 6 entry nodes are used.

Summary. These results indicate that (1) the top-ranked nodes provided by DEPIMPACT are effective in preserving critical edges and reducing non-critical edges when the top 2 entry nodes from the 3 types of system entities are used; (2) the mechanism that considers the types of system entities when choosing the top nodes achieves more stable results for different type of attacks than the one without considering the types of system entities.

D. RQ3: Revealing Attack Entries

The evaluation results in Section V-C show that ranking attack entries at the top is critical for DEPIMPACT to achieve good performance (very low M_{miss} and low M_{remain}). 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 size

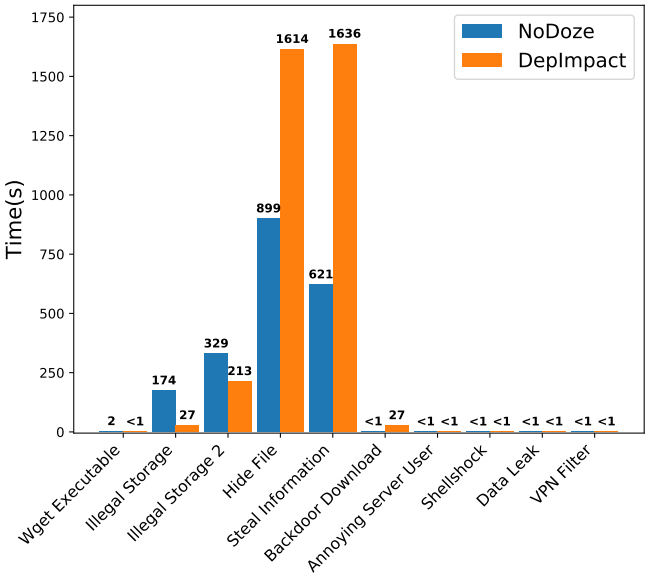

Fig. 6: Runtime performance of DEPIMPACT and NoDoze

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 computer the dependency weight.

Table VI 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 3.13) 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 48.69%, 46.95%, 29.34% and 99.92% improvement in ranking the attack entries.

The comparison among DEPIMPACT, DEPIMPACT--, and DEPIMPACT- demonstrates the effectiveness of our three selected features, and the comparison between DEPIMPACT and the average-projection approach demonstrates the superiority of our discriminative feature projection scheme over the fixed average projection for computing the dependency weights.

E. RQ4: System Performance

To understand the performance of DEPIMPACT, we measure the execution time of each step in DEPIMPACT, as shown in Table VII. On average, DEPIMPACT takes $430s$ to finish analyzing an attack, and dependency graph construction (i.e., Causality Analysis) requires $69.98s$ and edge merge requires $8.56s$. 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 VII, we observe that (1) DEPIMPACT takes more time for dependency weight computation ($\sim 102s$) because DEPIMPACT uses the Multi-KMeans++ clustering and LDA to find the optimal projection

TABLE VII: Runtime performance of DEPIMPACT and baseline approach

Attack	Causality Ana.(s)	Edge Merge(s)	Dependency Weight Computation (s)		Dependency Impact Propagation (s)	
			DEPIMPACT	Avg. Proj.	DEPIMPACT	Avg. Proj.
Wget executable	120.97	0.05	0.26	0.02	0.06	0.06
Illegal storage	92.86	0.38	7.43	0.39	19.48	47.77
Illegal storage 2	95.13	3.02	52.68	33.79	160.08	1,038.55
Hide File	223.63	42.16	463.68	16.14	1,150.35	8,486.32
Steal Information	129.82	39.51	479.02	15.98	1,157.45	8,128.28
Backdoor Download	19.74	0.44	13.87	0.32	12.75	24.05
Annoying Server User	17.23	0.01	0.18	0.01	0.03	0.03
Shellshock	0.05	0.03	0.07	0.01	0.02	0.06
Dataleak	0.09	0.01	0.28	0.02	0.14	0.16
VPN Filter	0.28	0.04	0.35	0.03	0.11	0.14
Avg.	69.98	8.56	101.78	6.67	250.05	1,772.54

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 80.23% 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. Figure 6 shows the results. We observe that DEPIMPACT is more efficient than NoDoze for 2 attacks (i.e., the "Illegal Storage" attack and "Illegal Storage 2" attack), as efficient as NoDoze for 5 attacks, and less efficient for 3 attacks. 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 (~ 800 edges) than NoDoze ($> 20,000$ edges). On average, DEPIMPACT needs 351.8s to finish the dependency weight computation and the dependency impact propagation, and NoDoze needs 202.65s to finish the anomaly score computation ($430.34s$ v.s. $281.19s$ 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 at the cost of poor graph reduction.

VI. DISCUSSION

A. Evasion Attacks

Existing causality analysis techniques, such as NoDoze [18], leverage execution profiles and reputations of entities (e.g., IP and file reputations) to identify anomaly edges. As shown in Section V-B, 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 need to compromise or cooperate with commonly used system processes for their attack steps, e.g., data copying. For example, the attackers can have their malicious payload copied over many times by a commonly used system process, such as `cp`, before writing to a malicious script. Then, to hide these copies, the attacker can control the same process to copy some garbage data to another irrelevant processes with larger data amount, and make this irrelevant process write data to impact the malicious script in the POI event. To do so, the attackers will have to compromise or cooperate with all other processes along the way to the POI event, which is very likely to trigger other alerts. Even if the attackers succeed in all these steps without being detected, we can still mitigate such attacks by applying DEPIMPACT on each of the write event (i.e., all the small writes of the malicious payload and the write of the garbage data) to the malicious script, and correctly identifying the attack sequences for the writes of the malicious payloads. We may also verify the integrity of the commonly used processes (e.g., checking control-flow integrity [53], [54]) or other process-based anomaly detection techniques [55], [56] to help distinguish these writes.

Alternatively, the attackers may use a set of different processes to accomplish their goals, and we can use rule-based techniques to limit or forbid unknown processes for moving data around the system. In any case, DEPIMPACT significantly raises the bar for the attackers to hide their attack steps.

B. Design Alternatives

For feature extraction, besides the features proposed, the design of 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 [57] could be a substitution for KMeans clustering, and there could be alternatives for LDA to achieve discriminative dimensionality reduction [27], [58]. We plan to explore these options in future work.

C. Runtime Performance Improvement

The performance of DEPIMPACT may benefit from database optimization and parallelization. Backward and forward causality analyses can be improved by adopting the database

optimization techniques to speed up the search [14], [15], 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 [59], [60]. We plan to explore these ideas in future work.

D. Multi-Host Attack Investigation

DEPIMPACT accepts a backward dependency graph produced by causality analysis and identifies the critical component of the graph. Attack steps on multiple hosts can be analyzed by performing cross-host causality analysis using existing techniques [9], [13], which produce causality graphs that include special network connection edges to represent connections among multiple hosts. These connection edges ensure that the dependency flows are not cut off across hosts. DEPIMPACT can directly take these dependency graphs as input and work with them seamlessly. In particular, in our evaluations, our multi-step intrusive attacks in Section V-A2 involve communications between two machines, C2 server and the victim machine. To investigate the attack on the victim machine, we used the system auditing events on the victim machine for evaluation.

VII. 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. [8], [9] 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. [16] 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 [12], [19], [24], [61], [62], prioritizing dependencies [13], [18], customized kernel [38], and optimizing storage [25], [26], [33], [63]. However, these techniques suffer from adoption limitations as they mainly rely on heuristic rules that cause loss of information [8], intrusive system changes such as binary instrumentation [12], [19] and kernel customization [38], or execution profiles that have limited generalizations [18]. 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. [14], [15] 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. [64] 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 [65]. Pasquier et al. [66] propose a runtime analysis of provenance by combining runtime kernel-layer reference monitor with a query module mechanism. Hossain et al. [67] 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 [28], 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 [68], and identifies spam pages as those with low scores. Similar ideas have been applied in security and privacy application scenarios including Sybil detection [69]–[71], fake review detection [72], and attribute inference attacks [73]. DEPIMPACT is the first work that applies the score propagation idea in system audit logging domain, with the focus on reducing the size of dependency graph and identifying relevant dependencies to facilitate forensic investigation.

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++ [45], 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) [27], which finds a linear combination of features that characterizes or separates multiple classes of objects.

VIII. 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 $\sim 2,700\times$ smaller graphs) while preserving the attack-relevant dependencies.

REFERENCES

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

[2] "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.

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

[4] "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.

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

[6] "The Equifax data breach," https://www.ftc.gov/equifax-data-breach.

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

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

[9] S. T. King, Z. M. Mao, D. G. Lucchetti, and P. M. Chen, "Enriching intrusion alerts through multi-host causality," in *NDSS*, 2005.

[10] X. Jiang, A. Walters, D. Xu, E. H. Spafford, F. Buchholz, and Y.-M. Wang, "Provenance-aware tracing of worm break-in and contaminations: A process coloring approach," in *ICDCS*, 2006.

[11] S. Ma, J. Zhai, Y. Kwon, K. H. Lee, X. Zhang, G. F. Ciocarlie, A. Gehani, V. Yegneswaran, D. Xu, and S. Jha, "Kernel-supported cost-effective audit logging for causality tracking," in *USENIX ATC*, 2018.

[12] Y. K., F. W., W. W., K. H. L., W. L., S. M., X. Z., D. X., S. J., G. F. C., A. G., and V. Y., "MCI : Modeling-based causality inference in audit logging for attack investigation," in *NDSS*, 2018.

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

[14] P. Gao, X. Xiao, Z. Li, F. Xu, S. R. Kulkarni, and P. Mittal, "AIQL: Enabling efficient attack investigation from system monitoring data," in *USENIX ATC*, 2018.

[15] P. Gao, X. Xiao, D. Li, Z. Li, K. Jee, Z. Wu, C. H. Kim, S. R. Kulkarni, and P. Mittal, "SAQL: A stream-based query system for real-time abnormal system behavior detection," in *USENIX Security*, 2018.

[16] A. Goel, K. Po, K. Farhadi, Z. Li, and E. de Lara, "The taser intrusion recovery system," in *SOSP*, 2005.

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

[18] W. U. Hassan, S. Guo, D. Li, Z. Chen, K. Jee, Z. Li, and A. Bates, "Nodoze: Combatting threat alert fatigue with automated provenance triage," 2019.

[19] S. Ma, X. Zhang, and D. Xu, "Protracer: towards practical provenance tracing by alternating between logging and tainting," 2016.

[20] G. P. Spathoulas and S. K. Katsikas, "Reducing false positives in intrusion detection systems," *Computers & Security*, vol. 29, no. 1, pp. 35–44, 2010.

[21] T. Pietraszek, "Using adaptive alert classification to reduce false positives in intrusion detection," in *Recent Advances in Intrusion Detection*, E. Jonsson, A. Valdes, and M. Almgren, Eds. Springer Berlin Heidelberg, 2004, pp. 102–124.

[22] W. U. Hassan, M. Lemay, N. Aguse, A. Bates, and T. 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.

[23] A. Kharraz, S. Arshad, C. Mulliner, W. K. Robertson, and E. Kirda, "UNVEIL: A large-scale, automated approach to detecting ransomware," in *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, T. Holz and S. Savage, Eds., 2016, pp. 757–772.

[24] K. H. Lee, X. Zhang, and D. Xu, "High accuracy attack provenance via binary-based execution partition," in *NDSS*, 2013.

[25] Z. Xu, Z. Wu, Z. Li, K. Jee, J. Rhee, X. Xiao, F. Xu, H. Wang, and G. Jiang, "High fidelity data reduction for big data security dependency analyses," in *CCS*, 2016.

[26] Y. Tang, D. Li, Z. Li, M. Zhang, K. Jee, X. Xiao, Z. Wu, J. Rhee, F. Xu, and Q. Li, "Nodemerge: Template based efficient data reduction for big-data causality analysis," in *ACM CCS*, 2018.

[27] S. Mika, G. Ratsch, J. Weston, B. Scholkopf, and K.-R. Muller, "Fisher discriminant analysis with kernels," 1999.

[28] Z. Gyöngyi, H. Garcia-Molina, and J. Pedersen, "Combating Web Spam with Trustrank," in *Proceedings of the International Conference on Very Large Data Bases*, 2004.

[29] E. Database, "Exploit Database," 2017, https://www.exploit-db.com/.

[30] Y. Kwon, F. Wang, W. Wang, K. H. Lee, W.-C. Lee, S. Ma, X. Zhang, D. Xu, S. Jha, G. F. Ciocarlie *et al.*, "Mci: Modeling-based causality inference in audit logging for attack investigation." in *NDSS*, 2018.

[31] "cyberkillchain," https://www.lockheedmartin.com/en-us/capabilities/cyber/cyber-kill-chain.html.

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

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

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

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

[36] B. Cantrill, A. Leventhal, and B. Gregg, "DTrace," 2017, http://dtrace.org/.

[37] Sysdig, "Sysdig," 2017, https://sysdig.com/.

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

[39] S. A. Crosby and D. S. Wallach, "Efficient data structures for tamper-evident logging," in *USENIX Security Symposium*, 2009, p. 317–334.

[40] D. A. Wagner and P. Soto, "Mimicry attacks on host-based intrusion detection systems," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2002, pp. 255–264.

[41] M. Bishop, *Introduction to Computer Security*. Addison-Wesley Professional, 2004.

[42] C. Kruegel, F. Valeur, and G. Vigna, *Intrusion Detection and Correlation - Challenges and Solutions*, ser. Advances in Information Security. Springer, 2005, vol. 14.

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

[44] J. Friedman, T. Hastie, and R. Tibshirani, *The elements of statistical learning*. Springer series in statistics New York, NY, USA:, 2001, vol. 1, no. 10.

[45] D. Arthur and S. Vassilvitskii, "K-means++: The advantages of careful seeding," in *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '07, 2007.

[46] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Technical Report 1999-66, November 1999, previous number = SIDL-WP-1999-0120.

[47] FireEye Inc., "HammerToss: Stealthy Tactics Define a Russian Cyber Threat Group," FireEye Inc., Tech. Rep., 2015.

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

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

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

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

[52] S. Shevchenko, "VPNFilter 'botnet': a SophosLabs analysis," *A SophosLabs technical paper*, 2018.

[53] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, pp. 1–40, 2009.

[54] M. Khandaker, W. Liu, A. Naser, Z. Wang, and J. Yang, "Origin-sensitive control flow integrity," in *USENIX Security Symposium*, 2019, pp. 195–211.

[55] S. Sivakorn, K. Jee, Y. Sun, L. Kort-Parn, Z. Li, C. Lumezanu, Z. Wu, L. Tang, and D. Li, "Countering malicious processes with process-dns association," in *Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2019.

[56] S. Wang, Z. Chen, X. Yu, D. Li, J. Ni, L. Tang, J. Gui, Z. Li, H. Chen, and P. S. Yu, "Heterogeneous graph matching networks for unknown

malware detection," in *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence (IJCAI)*, 2019, pp. 3762–3770.
[57] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Comput. Surv.*, vol. 41, no. 3, pp. 15:1–15:58, Jul. 2009. [Online]. Available: http://doi.acm.org/10.1145/1541880.1541882
[58] M. Sugiyama, "Local fisher discriminant analysis for supervised dimensionality reduction," in *Proceedings of the 23rd international conference on Machine learning*. ACM, 2006.
[59] D. Gleich, L. Zhukov, and P. 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*, vol. 13, p. 22, 2004.
[60] C. Kohlschütter, P.-A. Chirita, and W. Nejdl, "Efficient parallel computation of pagerank," in *European Conference on Information Retrieval*. Springer, 2006, pp. 241–252.
[61] Y. Ji, S. Lee, E. Downing, W. Wang, M. Fazzini, T. Kim, A. Orso, and W. 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*. ACM, 2017, pp. 377–390.
[62] Y. Ji, S. Lee, M. Fazzini, J. Allen, E. Downing, T. Kim, A. Orso, and W. Lee, "Enabling refinable cross-host attack investigation with efficient data flow tagging and tracking," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 1705–1722.
[63] M. N. Hossain, J. Wang, R. Sekar, and S. D. Stoller, "Dependence-preserving data compaction for scalable forensic analysis," in *USENIX Security Symposium*, 2018, pp. 1723–1740.
[64] S. M. Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. Venkatakrishnan, "HOLMES: real-time APT detection through correlation of suspicious information flows," 2019.
[65] S. M. Milajerdi, B. Eshete, R. Gjomemo, and V. Venkatakrishnan, "Poirot: Aligning attack behavior with kernel audit records for cyber threat hunting," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019, p. 1795–1812.
[66] T. Pasquier, X. Han, T. Moyer, A. Bates, O. Hermant, D. Eyers, J. Bacon, and M. Seltzer, "Runtime Analysis of Whole-system Provenance," in *ACM CCS*. ACM, 2018.
[67] M. N. Hossain, S. M. Milajerdi, J. Wang, B. Eshete, R. Gjomemo, R. Sekar, S. D. Stoller, and V. N. Venkatakrishnan, "SLEUTH: real-time attack scenario reconstruction from COTS audit data," in *USENIX Security Symposium*, 2017, pp. 487–504.
[68] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank Citation Ranking: Bringing Order to the Web," Technical Report 1999-66, 1999.
[69] Q. C., M. S., X. Y., and T. P., "Aiding the detection of fake accounts in large scale social online services." USENIX, 2012.
[70] N. Z. Gong and D. Wang, "On the security of trustee-based social authentications," *Trans. Info. For. Sec.*, vol. 9, no. 8, 2014.
[71] P. Gao, B. Wang, N. Z. Gong, S. R. Kulkarni, K. Thomas, and P. Mittal, "Sybilfuse: Combining local attributes with global structure to perform robust sybil detection," in *CNS*, 2018.
[72] S. Rayana and L. Akoglu, "Collective opinion spam detection: Bridging review networks and metadata," in *KDD*, 2015.
[73] J. Jia, B. Wang, L. Zhang, and N. Z. Gong, "Attriinfer: Inferring user attributes in online social networks using markov random fields," in *WWW*, 2017.