REPTRACKER: Towards Automatic Attack Investigation via Weighted Causality Analysis

***Abstract*—Causality analysis is an important solution for attack investigation, which analyzes the system auditing events about system calls and presents the dependencies among system auditing events as a dependency graph, with nodes being system entities (*e.g.,* files and network sockets) and edges being dependencies (*e.g.,* file read/write). Such a tool is especially prominent for investigating Advanced Persistent Threat (APT) attacks that are usually conducted in multiple stages by exploiting various vulnerabilities. However, causality analysis suffers from dependency explosion and requires non-trivial manual efforts to inspect the dependency graph for surfacing attack provenance. We propose a weighted causality analysis approach, REPTRACKER, which (i) assigns weights to the dependencies to distinguish critical dependencies and non-critical dependencies, and (ii) propagates reputation values from chosen seed nodes to suspicious nodes to automate the process of graph inspection. The evaluations on a wide range of cases for key system interfaces and real APT attacks demonstrate the effectiveness of REPTRACKER in graph reduction based on computed weights and automated graph inspection based on reputation propagation.**

I. INTRODUCTION

Advanced Persistent Threat (APT) attacks [18], [47] have caused many well-protected companies with significant financial losses [5], [16], [17], [42], [50]. These APT attacks often infiltrate into target systems in multiple stages and span a long duration of time with a low profile, posing challenges for detection and investigation. Thus, enterprises have a strong need for solutions that can unfold these attacks for identifying root causes and ramifications, which is critical to perform system recovery in a timely manner and prevent future compromises.

Causality analysis [27], [29]–[31], [37] has emerged as an important solution for investigation of APT attacks. It is built upon *ubiquitous system monitoring* that collects system level auditing events about system calls. Causal analysis assumes causal dependencies between system entities (*e.g.,* processes) and system resources (*e.g.,* files and network sockets) involved in the same system call event (*e.g.,* a process reading a file), and organizes these system call events as a *dependency graph*, where nodes represent system entities/resources, edges represent dependency among the nodes, and the direction of the edges represent data flow. By inspecting such a dependency graph, security analysts can trace back POI (Point-Of-Interest) events to identify the entry point of the attacks and understand the damages of the attacks. For example, when an attacker exploits vulnerabilities for installing and executing malicious payloads, the dependency graph generated by causality analysis can reveal such vulnerable interfaces that accept malicious inputs from the network. Such capability is particularly important in attack investigation, since the causality of malicious events reveal the attack provenance.

However, there are two major limitations of causality analysis. First, causality analysis suffers from dependency explosion [33], [36]. For example, a long-running process that accepts many inputs (*e.g.,* a web browser or a file manager) can cause its output events (*e.g.,* writing to a file) to have dependencies on all the preceding input events. As such, the generated dependency graph for a suspicious downloaded file often consists of hundreds or even thousands of nodes and tens of thousands of edges, and the *critical edges* that reveal the attack provenance are buried in a huge number of irrelevant edges. Second, while a dependency graph already significantly reduces the efforts in revealing the attack provenance, the inspection of the graph is still a labor-intensive work for the security analysts, especially when the graph size is often large. Thus, it is a daunting task for the security analysts to inspect dependency graphs manually and identify the critical edges.

Existing works have mainly focused on addressing the dependency explosion problem via data reduction [24], [28]–[30]. In particular, these works propose to remove irrelevant dependencies via (i) heuristically pruning specific files [29], [30], (ii) collecting enhanced run-time information through binary instrumentation [31], [35], customized kernel [11], and taint analysis [38], or (iii) prioritizing dependencies using reference models of normal activities [36]. However, heuristics often cause the loss of important dependencies [29], [51]; changing end-user systems, such as instrumentation and customization on kernel, are not practical in many organizations such as industry or government; and reference models are difficult to control and they cannot be easily generalized from one organization to the other. Furthermore, none of these works provide tool support for the inspection of the output graph, and they still require non-trivial manual efforts to reveal the attack provenance.

In this paper, we propose a weighted causality analysis approach, REPTRACKER, which (i) assigns weights to edges in the dependency graph for surfacing critical edges, and (ii) propagates reputation values on the weighted dependency graph to automatically determine whether the system entities involved in the POI events have similar reputations as the trusted sources (*e.g.,* verified binaries and trusted websites) or malicious sources (*e.g.,* USB sticks or unknown IPs). To address dependency explosion, REPTRACKER allows security analysts to hide non-critical edges without pruning critical edges by adjusting a threshold values within the suggested range. To provide tool support for automatic inspection of dependency graph, the reputation propagation of RepTracker can automatically determine whether the POI events in the dependency graph are malicious payloads (*e.g.,* executable files coming from malicious tools or suspicious websites) or benign entities (*e.g.,* files produced by trusted processes).

There are two key insights of RepTracker. First, given a dependency graph of a POI event, its critical edges often possess different properties than non-critical edges that are unlikely to reveal attack provenance. For example, critical edges often read or write some files with the data amounts close to the file involved in the POI event (referred to as *target file*), and the time stamps associated with these edges are quite close to the time stamp of the POI event.

Based on this insight, RepTracker extracts three novel types of discriminative features from a system auditing event: (i) **relative data amount difference** that measures the distance between the number of bytes of data processed by the system call (*e.g.,* read 1k bytes) and the target file's size; (ii) **relative time difference** that measures the distance of the time stamps between a dependency event and the POI event; (iii) **concentration degree** that measures the ratio of incoming edges over outgoing edges for a system entity, which is particularly useful for smoothing out the impacts of libraries with no incoming edges and long-running processes with many outgoing edges. Based on these features, RepTracker uses a novel *discriminative local feature projection* mechanism to combine three features into a single weight for each edge (details in Section IV-D).

The second key insight of RepTracker is that a file is very likely to contain malicious payloads if it is created by a malicious file or comes from a malicious website. On the other hand, a file can be trusted if it is created by official installation files (*e.g.,* official Microsoft installation packages and Google updater) or other trusted sources.

Based on this insight, RepTracker assigns seed nodes with initial reputations and propagates the reputations from the seed nodes to all other nodes. Seed nodes are usually trusted sources like Chrome updater (assigned with high reputations), system libraries like libc (assigned with neutral reputations), or unknown sources like USB sticks or malicious websites (assigned with low reputations). A node in the dependency graph receives its reputation by aggregating the weighted reputations from all of its parent nodes. The propagation process is an iterative process and repeats until the reputation of all the nodes become stable.

We evaluate RepTracker by constructing 21 cases from operations that employ key system interfaces (11 cases) and five real APT attack scenarios (10 cases) Our graph reduction results show that RepTracker is able to effectively reduce up to 72.8% nodes and 97.4% edges in the original dependency graph, thanks to its causality reduction and edge merge components. We also show that with the help of weighted dependency graph produced by RepTracker, security analysts are able to effectively surface critical edges from non-critical edges by controlling the threshold on filtering edge weights. Our reputation propagation results show that RepTracker is able to accurately propagate the reputation from trusted seeds in both high initial seed reputation setting (*HighRP*) and low initial seed reputation setting (*LowRP*), thanks to its novel weight computation mechanism. Compared to the baseline approaches, the local cluster and projection approach employed by RepTracker achieves the best performance for most of the cases in both *HighRP* setting and *LowRP* setting. Specifically, RepTracker achieves up to 22% average percentage improvement in *HighRP* setting and 92% average percentage improvement in *LowRP* setting.

TABLE I: Representative attributes of system entities

Entity	Attributes	Shape in Graph
File	Path	Ellipse
Process	PID and Name	Square
Network Connection	IP and Port	Parallelogram

Below are the main contributions of RepTracker:

- RepTracker is a novel causality analysis approach that addresses the shortcomings of the existing approaches.
  - RepTracker extracts the features from the system call event, and hence it does not require changes of end-user system.
  - RepTracker leverages the differences of the extracted features to compute weights for distinguishing critical and non-critical edges, and it does not need a reference model of normal activities.
  - RepTracker hides non-critical edges with low weights, and thus does not use heuristics that can cause a loss of dependencies for certain entities.
- RepTracker provides reputation propagation to automate the inspection of the dependency graph, which has limited support from the existing approaches.
- RepTracker is evaluated on representative cases for key system interfaces that are vulnerable for attacks and real APT attacks, and the results demonstrate the effectiveness of RepTracker in addressing dependency explosion and propagating reputation.

II. Background and Motivating Example

A. System Monitoring

System monitoring collects auditing events about the system calls that are crucial in security analysis, describing the interactions between system entities and resources. As shown in previous studies [22]–[24], [27], [29]–[31], [37], on mainstream operating systems (Windows, Linux, and Mac OS), system entities in most cases are files, network connections, and processes [24], [27], [29], [30], and the collected system calls are mapped to three major types of system events: (i) process creation and destruction, (ii) file access, and (iii) network access. As such, we consider *system entities* as *files*, *processes*, and *network connections*. We define an interaction among entities as an *event*, which is represented using the triple $\langle$*subject, operation, object*$\rangle$.

Entities and events have various attributes (Tables I and II). The attributes of an entity include the properties to describe the entities (*e.g.,* file name, process name, and IP address), and

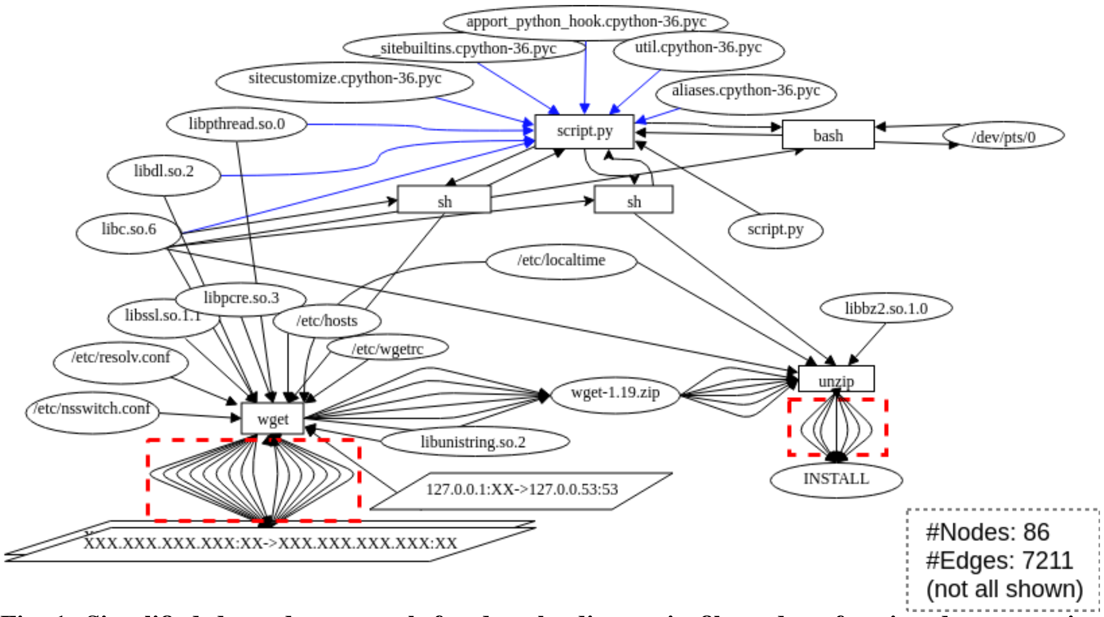

Fig. 1: Simplified dependency graph for downloading a zip file and performing decompression

TABLE II: Representative attributes of system events.

Operation	Read/Write, Execute, SendTo/Recvmsg
Time	Start Time/End Time, Duration
Misc.	Subject, Object, Failure Code

the unique identifiers to distinguish entities (*e.g.,* file path and process ID). The attributes of an event include event origins (*i.e.,* start time/end time), operations (*e.g.,* file read/write), and other security-related properties (*e.g.,* system call return code).

B. Causality Analysis

Causality analysis analyzes the auditing events to infer the dependencies among system entities and present the dependencies as a directed graph. In the dependency graph $G(E, V)$, a node $v \in V$ represents a process, file or network socket. An edge $e = (u, v) \in E$ indicates a system auditing event involving 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 data flow. Each edge is associated with a time window, $tw(e)$. We use $ts(e)$ and $te(e)$ to represent the starting time and the ending time of e. Formally, in the dependency graph, for two event edges $e_1 = (u_1, v_1)$ and $e_2 = (u_2, v_2)$, there exist causality 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: (i) *backward causality analysis* and (ii) *forward causality analysis*. 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 causality dependencies on u, and a forward causality analysis finds all events on which v have causality dependencies. Backward causality analysis can help identify the entry points of attacks and forward causality analysis can help investigate the ramification of attacks.

C. Motivating Example

Figure 1 shows an example dependency graph for a representative system task: a process `script.py` executes `wget` to download a file named `wget-1.19.zip`, and unzips it to obtain the file `INSTALL`. Such task is often seen in malicious payloads [12]. The POI event of the graph is the event that creates the file `INSTALL`. The original dependency graph obtained by applying causality analysis on the POI event contains 86 nodes and 7211 edges. Due to space limit, only relevant part is shown in the figure.

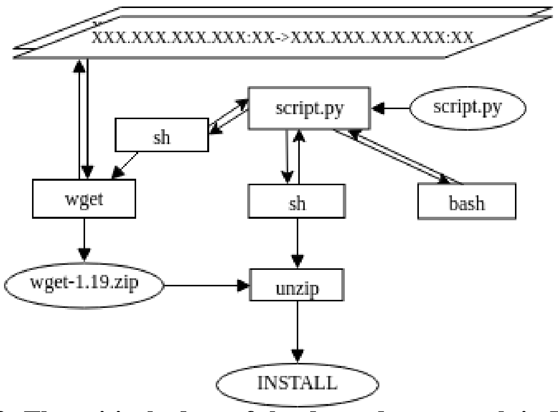

Fig. 2: The critical edges of the dependency graph in Fig. 1

Surfacing Critical Edges. The first goal of attack investigation using dependency graph is to surface attack provenance, *i.e.,* finding critical edges. Figure 2 shows the 14 critical edges of the example dependency graph. Obviously, it requires non-trivial efforts to search these critical edges from the huge dependency graph with 86 nodes and 7211 edges, and there is a strong need to automate this identification process.

However, causality analysis [29], [30] typically uses time windows to make decisions on whether an edge has causal dependency on the POI event, which contains limited information to distinguish critical edges. For example, the `script.py` process node has many incoming edges, but the edges marked in blue are coming from system libraries, which give little information on surfacing the attack provenance, while the edges that connect to the process `sh` show much more useful information for investigating the task. But all these edges cannot be pruned since the end times of their time windows are all before the POI event.

To address this challenge, RepTracker leverages the insight that critical edges often possess properties that are different from non-critical edges: (i) the time difference between the critical edges and the POI event is usually small; (ii) the data amount processed by the critical edges is similar to the target file in the POI event; (iii) source nodes of non-critical edges have either no incoming edges (*e.g.,* system libraries) or many outgoing edges (*e.g.,* long-running processes), while the source node of the critical edges usually has only a few incoming edges and only a few outgoing edges (*i.e.,* "high concentration"). To capture these properties, RepTracker computes three novel features for each edge: *relative time difference*, *relative data amount difference*, and *concentration degree* (details in Section IV), and use them to determine the weight for each edge, producing a weighted dependency graph.

Note that with the weights on the edges, security analysts can control the threshold value on hiding some non-critical edges with low weights when identifying the critical edges. But when the security analysts need to look into some part of the graph with more details, he may lower the threshold value and the non-critical edges that contain more detailed information (*e.g.,* library files) will be shown.

Reputation Propagation. The second goal of attack investigation using dependency graph is to determine whether the file `INSTALL` contains malicious payloads. If we look at the critical edges, we can trace back from `INSTALL` to `script.py`, which is the creator of `INSTALL`. This indicates that if `script.py` is suspicious, `INSTALL` is very likely to contain malicious payloads; otherwise, `INSTALL` is likely to be a benign installation script.

To automate the process in inspecting the dependency graph for determining whether a POI event contains malicious payloads, RepTracker performs reputation propagation on the weighted dependency graph. In this example, if we assume `script.py` is suspicious (*i.e.,* 0), and all the system libraries with a neural reputation (*i.e.,* 0.5). Then RepTracker propagates the reputation from these seed nodes through all the edges, where the weights of the edges are proportional to the reputation being propagated from the nodes. Thus, if the weights of the critical edges are significantly higher than the weights of the non-critical edges, the contribution from `script.py` to `wget` dominates the contributions from system libraries to `wget`. This will cause `INSTALL` to have a reputation close to 0, which is similar to `script.py`, but not the system libraries. In this way, RepTracker can automatically determine that `INSTALL` is very likely to contain malicious payloads.

III. Overview and Threat Model

Figure 3 shows the work flow of RepTracker. RepTracker consists of three phases: (1) dependency graph generation, (2) graph preprocessing & weight assignment, and (3) attack investigation. In the first phase, RepTracker leverages system auditing tools such as Auditd [44], ETW [39], DTrace [13], and Sysdig [48] to collect system level auditing events about system calls. Given a POI event, RepTracker parses the collected events and applies causality analysis [29], [30] to obtain the dependency graph for the event. In the second phase, RepTracker performs graph preprocessing to merge the same type of edges between two nodes and split the nodes to remove parallel edges between nodes. This process transforms the dependency graph into a simple directed graph that is easier for weight computation and reputation propagation. RepTracker then generates three types of features and computes the weights for each edge, producing a weighted dependency graph. In the third phase, RepTracker surfaces critical edges by providing a suggested range of threshold values to prune non-critical edges, and propagates reputations to identify malicious payloads.

Threat Model. RepTracker is a causality analysis tool over system monitoring data, and thus we follow the threat model of previous works on system monitoring data [11], [22], [23], [29], [30], [35], [36]. We assume that the system monitoring data collected from kernel space [39], [44] is not tampered, and that the kernel is trusted. Any kernel-level attack that deliberately compromises security auditing systems is beyond the scope of this work.

The attacker executes APT attacks involving multiple steps such as target discovery and data exfiltration. 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. The main goal of the attacker is to inject her malicious files into the victims system without being detected. In this work, we assume the attacker does not know how the proposed reputation system operates, and hence we do not consider the potential attacks against the reputation system.

TABLE III: Processed System Calls

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

IV. RepTracker Design

In this section, we present the three phases and the components of RepTracker in detail.

A. Phase I: Dependency Graph Generation

In this phase, RepTracker leverages system auditing tools (*e.g.,* Sysdig [48]) to collect information of system calls from the kernel, and then parses the collected events to build a global dependency graph. RepTracker focuses on three types of system events: (i) process creation and destruction, (ii) file access, and (iii) network access, and Table III shows the representative system calls for these three types of events in Linux. Particularly, 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

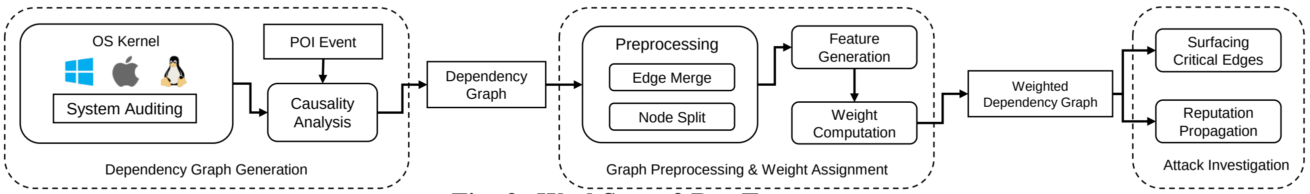

Fig. 3: Workflow of RepTracker

socket 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. Thus, we use 5-tuple (*⟨srcip, srcport, dstip, dstport, protocol⟩)* as a network socket's unique identifier. Failing to distinguish different entities causes problems in relating events to entities, especially for tracking dependencies of events. For each system call, RepTracker also extracts the event attributes as shown in Table II. RepTracker filters out failed system calls, which could cause false dependencies among events.

Algorithm 1: Backward Causality Analysis

```
Input: POI Event $e_s$, Global Dependency Graph $G$
Output: Dependency Graph for the POI event $G_d$
1 $G_d \leftarrow new\ Graph()$
2 $Queue.add(e_s)$
3 while $Queue$ is not Empty do
4     $v \leftarrow Queue.poll()$
5     $set \leftarrow G.incomingEdgeOf(v)$
6     for $e \in set$ do
7         if $ts(e) < te(v)$ then
8             $G_d.add(e)$
9             $Queue.offer(e.source)$
```

Causality Analysis. Given a POI event e_s, RepTracker applies causality analysis [29] to produce a dependency graph G_d, as shown in Algorithm 1[1]. Algorithm 1 adds e_s to a queue (Line 2), and repeats the process of finding eligible incoming edges of the edges (*i.e.,* incoming edges of the source nodes of edges) in the queue (Lines 3-9) until the queue is empty. The output of this module is a dependency graph only contains relevant system entities and events about the given POI event.

B. Phase II: Graph Preprocessing

RepTracker performs graph preprocessing to transform the dependency graph from a directed multigraph to a simple directed graph with no parallel edges.

Edge Merge. The dependency graph produced by causality analysis often have many edges between the same pair of nodes. For example, in Figure 1, the **wget** process has many edges going to and coming from a network socket, and the **unzip** process also has many edges going to and coming from the file **INSTALL**. Both of these pairs are shown in red dotted rectangles in the Figure 1.

The reason for generating these excessive edges is that the OS typically finishes a read/write task (*e.g.,* file read/write) by distributing the data to multiple system calls where each system call processes only a portion of the data. Inspired by the recent work that proposes CPR (Causality Preserved Reduction) [51] for dependency graph reduction, RepTracker merges the edges between two nodes. As shown in the study [51], CPR does not work well for processes that have many interleaved read and write system calls, which introduces excessive causality. As such, RepTracker adopts a more aggressive approach than CPR: for edges between two nodes that represent same system call (*e.g.,* file reads from **read** or network reads from **recvfrom**), RepTracker will merge them into one edge and use their merged time windows as the time window for the merged edge. Since such merge is performed after the dependency graph generation, all the dependencies are still preserved but only the time windows of certain edges are merged.

Node Split. After the edge merge, the dependency graph may still have parallel edges, *i.e.,*, edges indicating read or write from different system calls. For example, a process may receive data from a network socket via both **read** and **recvfrom**. These parallel edges create complications for weight computation and reputation propagation, making it difficult to represent the weights of all the edges using a transition matrix. Thus, RepTracker splits a node u into multiple nodes, where each node has only one outgoing edge pointing to node v.

RepTracker first finds all the pair of nodes that have parallel edges. For a pair of nodes (u, v) that have parallel edges pointing from u to v, RepTracker creates copies of u for each parallel edge. Each copy of u is mapped to one original outgoing edge to v, and inherits all u's incoming edges that has causal dependency on the outgoing edge. After the node split, the dependency graph becomes a simple directed graph without parallel edges, which allows RepTracker to use a transition matrix to present the weights of edges.

C. Phase II: Feature Generation

To compute the weights for edges, RepTracker generates three novel discriminative features from the dependency graph.

- **Relative Time difference:** For an edge $e(u, v)$, we measure the difference between its end time $te(e)$ and the end time of the POI event $te(e_s)$. The intuition is that the event that occurred closer to the POI event is more temporally related to the POI event, and hence should be associated with a higher feature value. Thus, we define the *time difference feature* $f_{T(e)}$ for the edge e as:

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

[1]Forward causality analysis can be implemented using a similar way.

where $t_{(e)}$ and t_{e_s} represent timestamp values (we use the event end time) of the event edge e and the POI event e_s.

In (1), $f_{T(e)}$ is a positive real number. The smaller the time difference $\mid t_e - t_{e_s} \mid$ gets, the larger the value of $f_{T(e)}$ becomes, and hence the more trust the source node u puts on the destination node v in the temporal dimension. Special action needs to be taken when the event edge e is actually the POI event, and hence $\mid t_e - t_{e_s} \mid = 0$. To handle such a case, we specify that the minimal time difference to be one tenth of the minimal unit (*i.e.,* nanosecond) for the timestamp field in the audit logging framework (*i.e.,* 1e−10), and use this minimal value when computing the time difference feature of the POI event. By doing so, the POI event will have the highest time difference feature value $f_{T(e_s)} = ln(1 + 1\text{e}{-10})$.

- **Relative Data Amount Difference:** For an edge $e(u, v)$, we measure the difference between the data amount associated with this event edge and the size of the target system entity in the POI event e_s. The intuition is that the closer the data amount of an event is to the size of the target system entity, the more likely this event edge is related to the POI event, and hence should be associated with a higher feature value. Thus, we define the *data amount difference feature* $f_{D(e)}$ for the edge e as:

$$f_{D(e)} = 1/(\mid s_e - s_{e_s} \mid + \alpha), \quad (2)$$

where s_e and s_{e_s} represent the data amount of the event edge e and the size of the POI file.

Note that in (2), we use a small positive number α to handle the special case when e is the POI event. Thus, the POI event will have the highest data amount difference feature value $f_{D(e)} = 1/\alpha$. Empirically, we set $\alpha = 1e - 4$.

- **Concentration Degree:** One important category of non-critical edges that often appear in a causal graph are events that access system libraries [49], [51]. These edges are often associated with considerable data amount and occur at various timestamps, and hence using only $f_{T(e)}$ and $f_{D(e)}$ is less effective in distinguishing critical edges from non-critical ones. To address this challenge, we observe that most system library nodes do not have any incoming edges and these system library nodes are source nodes in the corresponding edges. Also, many long-running processes have few incoming edges but many outgoing edges, and thus the weight of each outgoing edge should be smaller. Based on this observation, we define the *concentration degree* for the edge $e(u, v)$ as:

$$f_{C(e)} = InDegree(u)/OutDegree(u), \quad (3)$$

where $InDegree(u)$ and $OutDegree(u)$ represent the in-degree and out-degree of the source node u in $e(u, v)$.

Note that if u is a system library node, $InDegree(u)$ is likely to be zero, and hence $f_{D(e)} = 0$. For seed nodes, since they are very important in initiating the reputation propagation, we set their concentration degree to be 1.

Local Feature Normalization. Before using the features to compute weights, it is often a good practice to scale them in the same range. Globally scaling them using standard methods such as range normalization and standardization [9] does not intuitively make much sense in our specific setting, since a node is only affected by its parents but not by its children or siblings. Recognizing this, we propose a novel *local feature scaling* scheme: for an edge $e(u, v)$, we *locally* normalize its feature f by the sum of the feature of all incoming edges of node v. Thus, for node v, this scheme enables the features of all its incoming edges to be compared in the same scale. The higher the normalized feature gets, the more influence its corresponding parent node has on the child node v.

D. Phase II: Weight Computation

The next task is to compute the weight for each edge from its three locally normalized features. Formally, for each edge $e(u, v) \in E$, we compute its *weight*, $W_e \in [0, 1]$, which models the level of trust that the node u puts on the node v. The challenge is to make these weights "discriminative", so that critical edges will have high weights and non-critical edges will have low weights.

One approach to compute W_e is to adopt supervised machine learning to train a binary classifier from three features and use probabilistic class estimates as weights. Those this approach works for many domains, it has several fundamental limitations in our specific problem context. Supervised learning approaches often require large amount of training data and require the data exhibit consistent distribution patterns across training data and testing data. However, our problem context is highly specialized for specific POI events (in fact, two of our three features are computed using attributes from POI events). What's more, our problem context is highly imbalanced and the number of critical edges is usually small. This unique context makes (1) the trained classifier hardly generalize across cases that investigate different POIs, and (2) the classifier lack enough training data to boost its accuracy.

Recognizing the limitations in supervised learning, REPTRACKER adopts an idea from unsupervised learning to perform dimensionality reduction via linear projection. Linear projection is known for its high interpretability and low computational cost [20]. Formally, for an edge e, its weight W_e is computed by projecting its feature vector $\mathbf{f}_{(e)} = (f_{T(e)}, f_{D(e)}, f_{C(e)})$ onto a unit vector $\mathbf{w}_e = (w_{T(e)}, w_{D(e)}, w_{C(e)})$:

$$\begin{aligned} W_e &= \mathbf{f}_e \cdot \mathbf{w}_e \\ &= w_{T(e)} * f_{T(e)} + w_{D(e)} * f_{D(e)} \\ &\quad + w_{C(e)} * f_{C(e)} \end{aligned} \quad (4)$$

where $\left\| \mathbf{w}_{(e)} \right\|_2 = 1$.

To address the key challenge of making weights "discriminative", we propose a novel *discriminative local feature projection* scheme that leverages the idea from discriminant analysis [9]. Different from another popular dimensionality reduction method PCA, which finds the projection vector that maximizes the variance of projected samples, discriminant analysis searches for the projection that maximizes the class

Algorithm 2: Weight Computation

```
Input: Weighted Dependency Graph, $G$
Output: Weighted Dependency Graph, $G$
1  for $v \in G$ do
2      $Set \leftarrow G.incomingEdgeOf(v)$
3      $T_{total} \leftarrow \sum_{e \in Set} f_{T_e}$
4      $D_{total} \leftarrow \sum_{e \in Set} f_{D_e}$
5      $C_{total} \leftarrow \sum_{e \in Set} f_{C_e}$
6      for $e \in Set$ do
7          $f_{T_e} \leftarrow f_{T_e}/T_{total}$
8          $f_{D_e} \leftarrow f_{D_e}/D_{total}$
9          $f_{C_e} \leftarrow f_{C_e}/C_{total}$
10     $Cluster1, Cluster2 \leftarrow$Kmeans++(Set)
11     $(a_{Te}, a_{De}, a_{Ce}) \leftarrow$LDA(Cluster1, Cluster2)
12     for $e \in Set$ do
13         $W_e \leftarrow a_{Te} * f_{T_e} + a_{De} * f_{D_e} + a_{Ce} * f_{C_e}$
14     $W' \leftarrow \sum_{e \in Set} W_e$
15     for $e \in Set$ do
16         $W_e \leftarrow W_e/W'$
```

separation of projected samples, and linear discriminant analysis (LDA) [40] the most popular approach. Algorithm 16 shows the detailed steps of weight computation.

However, we are not able to directly applying standard LDA, due to the following reasons: (1) LDA requires labeld samples from different classes but we do not have labels for critical edges and non-critical edges; (2) Our problem context is highly localized since a node is only affected by its parents but not by its children or siblings. Globally computing the projection vector for all edges might lead to serious bias (Section V-C2); (3) Standard LDA does not assume the within-class scatter matrix to be singular. However, this may be often in our context since the edges are highly imbalanced.

Next, we describe how we address these challenges in extending the standard LDA.

1) Local Edge Clustering: To address the first challenge of lacking class labels, for each node, REPTRACKER *locally clusters* its incoming edges using their features, and produces two clusters (one for critical edges, one for non-critical edges). We can then treat the two clusters as two classes (though still lacking labels) and apply LDA. Specifically, we use Multi-KMeans++ clustering algorithm [10]. Based on KMeans, KMeans++ uses a different method for choosing the initial seeds 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 set $k = 2$ and $n = 20$.

2) Local Feature Projection: Instead of globally applying LDA to all edges, for each node, we compute the projection vector *locally* for its incoming edges, so that its critical edges can be maximally separated from its non-critical edges after projection. Formally, for node v, assume we have a set of 3-dimensional samples $\{x^{(1)}, x^{(2)}, \ldots, x^{(N)}\}$ representing its incoming edges, N_1 of which belong to cluster w_1, and N_2 to cluster w_2 ($N1+N2 = N$). The mean vector of each cluster is $\mu_1 = \frac{1}{N_1}\sum_{x \in w_1} x$, $\mu_2 = \frac{1}{N_2}\sum_{x \in w_2} x$. The between-cluster scatter matrix is defined as $S_b = (\mu_1 - \mu_2)(\mu_1 - \mu_2)^T$. The within-cluster scatter matrix is defined as $S_w = \sum_{x \in w_i}(x - \mu_i)(x - \mu_i)^T$. LDA (2-clusters) then finds a vector w to maximize the following Fisher criterion:

$$J(w) = \frac{w^T S_b w}{w^T S_w w} \tag{5}$$

In other words, we are looking for a projection where samples from the same cluster are projected very close to each other (denominator $w^T S_w w$) and, at the same time, the projected means of different clusters are farther apart as possible (numerator $w^T S_b w$).

Taking the derivative of $J(w)$ and equate it to zero, we get $S_b w = \lambda S_w w$. If S_w is non-singular, we can convert Equation (5) to a standard eigenvalue problem $S_w^{-1} S_b w = \lambda w$. Solving it yields

$$w^* = \operatorname{argmax} J(w) = S_w^{-1}(\mu_1 - \mu_2) \tag{6}$$

Equation (6) is not applicable when the inverse of S_w does not exist (*i.e.,* S_w is singular). However, in our problem context, such condition may happen pretty often, given that the number of incoming edges for most nodes is not large and the number of critical edges is even smaller (*i.e.,* highly imbalanced).

Recognizing such limitation, we extend the standard LDA by adding support for cases in which S_w is singular. Basically, when S_w is singular, we select the projection vector from the following candidates:

- $S_w^+(\mu_1 - \mu_2)$: S_w^+ is the Moore-Penrose [8] inverse of S_w. When S_w is non-singular, $S_w^+ = S_w^{-1}$.
- $\mu_1 - \mu_2$

For each candidate, we compute the numerator of Fisher criterion $w^T S_b w$ (we normalize w first), and selects the one that has a larger value.

Special Cases. For cases the node has only one incoming edges, we skip the clustering and projection process and directly set its final weight to 1.

3) Post-processing: LDA computes the projection vector that maximizes the cluster separation after projection, but it does not guarantee which cluster has higher projected values. In fact, the direction of the projection vector matters a lot and negating the direction will change the relative magnitude of the projected values of the two clusters.

Adjusting Projection Vector Direction. Our goal is to make the cluster that contains critical edges have higher projected values compared to the cluster that contains non-critical edges. Fundamentally, this problem is difficult to solve, since we don't have accurate labels for critical edges and thus do not actually know which cluster contains the critical edges. We propose to address this problem using following heuristics:

- If all three dimensions of the projection vector are non-positive, negate.
- If all three dimensions of the projection vector are non-negative, do not negate.

- If the projection vector has both negative dimensions and positive dimensions, negate by condition:
  - If one cluster has seed edges and one cluster doesn't, negate the projection vector when necessary to make sure the cluster that has seed edges has a higher projected mean. This is based on the insight that seed edges should have high weights.
  - If both clusters have seeds edges or neither has seed edges, negate the projection vector when necessary to make sure the cluster that has a smaller size (*i.e.,* contains fewer edges) has a higher projected mean. This is based on the insight that among all incoming edges of a node, there are usually less critical edges than non-critical edges.

Our experimental results on a wide range of realistic cases (Section V-C2) clearly demonstrate the effectiveness of our discriminative local feature projection scheme. After adjusting the projection vector direction, we normalize it, and compute the weight for each edge using Equation (4).

Scaling the Projected Weights to 0-1 Range. After computing the projected weights, one more post-processing step is needed: some edges might have negative projected values, which will cause problem for later local weight normalization (Section IV-D4). To address this, we scale the projected values to the range $[0, 1]$. We further add a small offset (we use one hundredth of the difference of the smallest value and the second smallest value) to each scaled value so that we only have positive scaled weights.

4) Local Edge Weight Normalization: Similar to local feature normalization, for an edge $e(u, v)$, we *locally* normalize its edge weight by the sum of edge weights of all incoming edges of node v:

$$W_e = W_e / \sum_{e' \in IncomingEdge(v)} W_{e'}, \tag{7}$$

After normalization, the weights of all event edges are in range $[0, 1]$ and the sum of weights of all incoming edges of any node is equal to 1 (except for nodes that have no parents). This completes the Phase II by producing a weighted dependency graph with discriminative weights to distinguish critical edges from non-critical edges, which is amenable to the next attack investigation phase.

E. Phase III: Attack Investigation

Surfacing Critical Edges. REPTRACKER leverages the weights in the dependency graph for surfacing critical edges. Since the weights computed by REPTRACKER aim to maximizes the differences between critical and non-critical edges, we can specify a minimum weight as threshold and hide the edges with weights below the threshold. For example, the weights of edges from the system libraries usually have lower weights than those reflect attack provenance, and by carefully choosing this threshold, REPTRACKER can filter out the system libraries while retaining the attack provenance.

Given that the weight distribution is different for each dependency graph, REPTRACKER uses a threshold value computed using $T_w * W_{avg}$, where W_{avg} represents the average weights in the dependency graph. Based on our results in Section V-C1, choosing T_w within $[0.15, 2.0]$ can prune most of the non-critical edges while retain the critical edges. In practical, security analysts can first hide the non-critical edges, and gradually show the non-critical edges with slightly higher weights by lowing the threshold.

Reputation Propagation. To perform reputation propagation, REPTRACKER assigns seed nodes with initial reputations firstly. Seed nodes are usually trusted sources such as official updates like Microsoft updater or Chrome updater (assigned with high reputations), system libraries like libc (assigned with neutral reputations), or unknown sources like USB sticks or malicious websites (assigned with low reputations). For each trusted seed, its initial reputation is assigned as 1.0, for each untrusted or unknown seed, its initial reputation is assigned as 0.0. The underlying assumption is that if a file or a program is downloaded or installed from a reputable source, it should be more reliable than the file downloaded from an unknown website or from an untrusted equipment. The reputation of files and programs from the reliable sources and channels should be higher than the reputation of those from unknown sources and channels. Thus, a file from the trusted seed will has a reputation closer to 1.0, otherwise it should closer to 0.0. The initial reputations of seed nodes representing system packages are set to 0.5 (in between the initial reputations of trusted seeds and untrusted seeds). Because the system packages will be used by both legitimate users and attackers, we cannot predict a node's nature from its causal relationship with the nodes representing system packages.

Based on the weight of each edge and chosen seeds, we iterative update the node's reputation except the chosen seeds. Any node's reputation can be computed as follows:

$$R_v = \sum_{e \in IncomingEdge(v)} R_{source(e)} * W_e, \tag{8}$$

where $IncomingEdge(v)$ represents the set of incoming edges of v, $source(e)$ represent the source node of e, $R_{source(e)}$ represents the reputation of $source(e)$, and W_e represents the weight of e. This process will not stop until the change of all the nodes' reputations is smaller than a given threshold (*e.g.,* δ). Empirically, we set $\delta = 1.0E - 13$.

Algorithm 3 shows the details of reputation propagation. A node v in the dependency graph receives its reputation by aggregating the weighted reputations from all of its parent nodes (Lines 7-9). In our context, such aggregation style of reputation propagation works better than the distribution style, which ensures the sum of reputations for v's children nodes to be equal to v's reputation. The reason is that the distribution style will make the reputation degrade quickly in a few hops, which does not work well for dependency graphs that often have many paths with many hops. The process of reputation propagation is an iterative process. For each iteration, the algorithm computes the sum of reputation differences for all

Algorithm 3: Reputation Propagation

```
Input: Weighted Dependency Graph, $G$
Output: Weighted Dependency Graph, $G$
while $\delta > thresold$ do
    $diff \leftarrow 0.0$
    for $\forall v \in G$ do
        if $v$ $is$ $seed$ then
            continue
        $Set \leftarrow G.incomingEdgeOf(v)$
        for $\forall e \in Set$ do
            $s \leftarrow e.source$
            $res \mathrel{+}= s.reputation * e.weight$
        $rep \leftarrow res$
        $diff \mathrel{+}= |v.reputation - rep|$
        $v.reputation \leftarrow rep$
        $\delta \leftarrow diff$
```

the nodes (Line 11). The propagation terminates when the differences between the current iteration and the previous iteration is smaller than a threshold (Line 1), indicating that the reputations for all the nodes become stable. Note that the reputations of the seed nodes remain unchanged (Lines 4-6).

V. Evaluation

A. Evaluation Setup

The evaluations are conducted on a laptop with an Intel(R) Core(R) i5-3210M CPU (2.5GHz), 8GB RAM running 64bit Ubuntu 18.04.1. We constructed 11 cases for key system interfaces that are vulnerable for attacks and 10 cases for key steps in real APT attacks. We evaluate the effectiveness of RepTracker in surfacing critical edges and propagating reputation from seeds to identify benign and malicious payloads.

B. Evaluation Cases Construction

1) Representative Cases for Key System Interfaces: We constructed 11 representative cases that employ the common system interfaces that vulnerable for attacks [12].

- File merge: *2File*, *3File*, *USB-merge* (files stored in a USB drive)
- Invoke shell script: *shell-script* (list all files in the Home folder and write the results to a file)
- File download: *curl*, *wget*, *shell-wget* (wget called by a shell script), *python-wget* (wget called by a Python script)
- File download then decompress: *shell-wget-unzip*, *python-wget-unzip*
- File transfer: *scp*

2) Key Steps in APT Attacks: We performed five APT attacks that capture the important traits of APT attacks depicted from the Cyber Kill Chain framework [4]. In total, we constructed 10 cases for key steps in the five APT attacks.

APT Attack 1: Zero-Day Penetration to Target Host. The scenario emulates the attacker's behavior who penetrates the victim's host leveraging previously unknown Zero-day attack. Zero-day vulnerabilities are attack vectors that previously unknown to the community, therefore allow the attacker to put their first step into their targets. In our case, we assume that `bash` binary in victim's host is outdated and vulnerable to shellshock [1]. The victim computer hosts web service that has CGI written as BASH script. The attacker can run an arbitrary command when she passes the specially crafted attack string as one of environment variable. Leveraging the vulnerability, the attacker runs a series of remote commands to plant and run initial attack by: (1) transferring the payload (*penetration-c1*), (2) changing its permission, and (3) running the payload to bootstrap its campaign (*penetration-c2*).

APT Attack 2: Password Cracking after Shellshock Penetration. After 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 EXIF metadata (*password-crack-c1*). The behavior is a common practice shared by APT attacks [7], [19] to evade the network-based detection system based on DNS blacklisting.

Using the IP, The malware connects to C2 host. 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 (*password-crack-c2*) and runs it against password shadow files (*password-crack-c3*).

APT Attack 3: Data Leakage after Shellshock Penetration. After lateral movement stage, the attacker attempts to steal all the valuable assets from the host. This stage mainly involves the behaviors of local and remote file system scanning activity, copying and compressing of important files, and transferring to its C2 host. The attacker scans the file system, scrap files into a single compress file and transfer it back to C2 host (*data-leakage*).

APT Attack 4: Command-line Injection with Input Sanitization Failures. Different from the previous shellshock case, a program may contain vulnerabilities introduced by developer errors and this also can be a initial attack vector that invites the attacker into their target systems. To represent such cases, we wrote an web application prototype that fails to sanitize inputs for a certain web request, hence allows Command line Injection attack. Our prototype service mimics the Jeep-Cherokee attack case [41] which implements a remote access using the conventional web service API that internally uses DBUS service to run the designated commands. Due to the developer mistake, the web service fails to sanitize the remote inputs, the attacker can append arbitrary commands followed by semi-colon(`;`). Leveraging this vulnerability, we can download backdoor program (*commend-injection-c1*) and collect sensitive data (*command-injection-c2*).

APT Attack 5: VPNFilter. We prototyped a famous IoT attack campaign; VPNFilter malware [6], which infected millions of dozens of different IoT devices exploiting a number of known or zero-day vulnerabilities [2], [3]. The attack's significance lies in how the malware operates during its lateral movement stage following its initial penetration. The campaign employs up-to-date hacker practiced to bypass conventional security solutions based on static blacklisting approaches and has an architecture to download plug-in payload on-demand, at run-time. We prototyped the malware referring to one of its sample for x86 architecture [45].

The VPNFilter 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 (*vpnfilter-c1*). It downloads the VPNFilter stage2 from the stage2 server, and runs it (*vpnfilter-c2*).

C. Evaluation Results

1) Graph Reduction Results: Table IV shows the reduction in the number of nodes and in the number of edges after causality analysis (Section IV-A) and edge merge (Section IV-B). As we can see, the reduction is significant: (1) In most of the cases, REPTRACKER achieves more than half of nodes reduced. Causality analysis helps trim up to 72.8% nodes on average. (2) In most of the cases, REPTRACKER achieves more than 95% of edges reduced. Edge merge helps trim up to 97.74% edges on average.

To surface critical edges, REPTRACKER uses a threshold to hide non-critical edges. To provide a guidance on selecting this threshold, we test the filtering performance by selecting an increasing multiple of average weight of the whole graph from 0 to 2 with a pace of 0.05. We define the *threshold* as the average weight of the whole graph magnified by a number T_w (*i.e.,* threshold multiplier). Figure 4 shows the average percentage of edges remaining of all cases after filtering. We observe a turning point at $T_w = 0.15$ and the number of remaining edges will remain stable below 20%. Higher thresholds can lead to more graph size reduction. However, if we choose the threshold too high, we will lose track of some of the critical edges. We define the *missing point* as the exact threshold multiplier that leads to the first critical edge loss(Table V). Figure 5 shows the cumulative distribution of missing points. We observe that: (1) Two cases (*command-injection-c2*, *data-leakage*) have extremely high missing points ($T_w > 200$); (2) 5 out of 21 cases lost critical edges at $T_w = 2$. However, in these 5 cases, 2 of them(*Shell-wget*,*penetration-c1*) already have less than 10 non-critical edges at missing point and 3 of them also have significant reduction in edge numbers(Table V). (3) A plateau exists before $T_w = 2$ at a rate of 24%. This indicate most of the cases have a missing point greater than $T_w = 2$, which proves the efficacy of our weights to differ critical edges from non-critical edges.

Given that setting $T_w = 0.15$ is enough to filter out more than 80% of the non-critical edges and 76% of the cases have $T_w > 2$. A good strategy would be examining the graph at $T_w = 2$ to grab a rough sense then tuning down to $T_w = 0.15$ to review details. To avoid critical edge miss in some situation, then going down to $T_w = 0$. Rather than directly examine the graph after Edge Merge, this will save a lot of daunting labor.

2) Reputation Propagation Results: We evaluate the effectiveness of REPTRACKER in identifying malicious and benign payloads through reputation propagation.

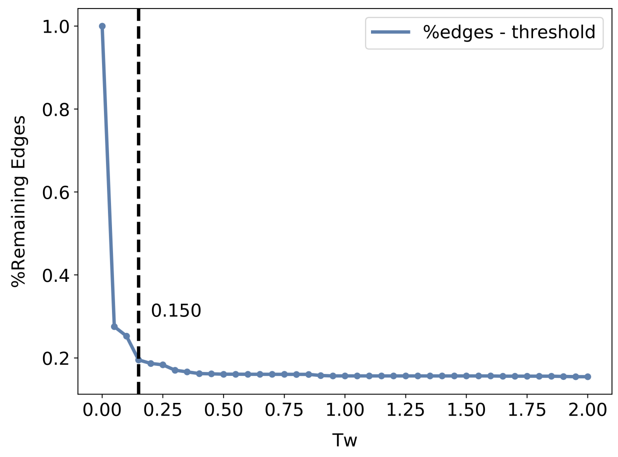

Fig. 4: Effectiveness of Filtering

The percentage of edges remaining after filtering drops significantly at $T_w = 0.15$ and remains stable below 20% (*i.e.,* filtering threshold equals T_w multiplies the average weight of all edges).

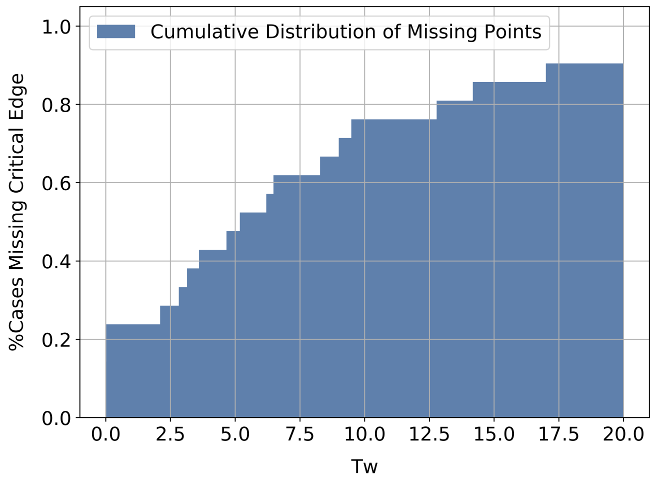

Fig. 5: Critical Edge Loss from Filtering

Missing points distribute mostly between $T_w = 2$ and $T_w = 18$. Note a plateau before $T_w = 2$.

Evaluation Setup. For each case, we set the initial reputation of seed nodes to both high seed reputation setting (*HighRP*; seeds RP = 1.0) and low seed reputation setting (*LowRP*; seeds RP = 0.0). For each setting and for all cases, we propagate the reputation from seed nodes according to Algorithm 3. We record the final reputation of POI nodes. We compare the POI reputation computed in the following four weight computation approaches:

- *ManualProjection*: We select a fixed parameter vector $(0.1, 0.5, 0.4)$ and normalize it to be a projection vector.
- *GlobalProjection*: We globally cluster all edges in the graph using Multi-KMeans++ and compute the projection vector using extended LDA.
- *GlobalProjectionNoOutlier*: Same as previous one, but for nodes that have only one incoming edge (*i.e.,* outlier edges), we do not consider these edges in the global clustering and projection vector computation, and directly assign their final

TABLE IV: Graph Reduction Result

Case	#N(Original)	#E(Original)	#N(Causality)	#E(Causality)	#N(Merge)	#E(Merge)	Node Ratio(%)	Node Reduction(%)	Edge Ratio(%)	Edge Reduction(%)
2File	75	2149	39	1900	39	38	52.00	48.00	1.77	98.23
3File	78	3131	40	2815	40	39	51.28	48.72	1.25	98.75
Python-wget	1210	9154	73	3265	73	82	6.03	93.97	0.90	99.10
Python-wget-unzip	154	7958	86	7211	86	101	55.84	44.16	1.27	98.73
Shell-script	38	527	19	42	19	22	50.00	50.00	4.17	95.83
Shell-wget	1161	9001	31	2685	31	37	2.67	97.33	0.41	99.59
Shell-wget-unzip	93	8506	41	7705	41	51	44.09	55.91	0.60	99.40
USB-merge	58	5916	8	4420	8	12	13.79	86.21	0.20	99.80
curl	126	3108	61	1681	61	64	48.41	51.59	2.06	97.94
scp	454	3711	58	109	58	74	12.78	87.22	1.99	98.01
wget	66	2457	28	2062	28	30	42.42	57.58	1.22	98.78
command-injection-c1	1702	6173	51	65	51	51	3.00	97.00	0.83	99.17
command-injection-c2	1702	6173	1164	3417	1164	1165	68.39	31.61	18.87	81.13
data-leakage	2303	76478	1809	59178	1809	1818	78.55	21.45	2.38	97.62
password-crack-c1	898	44026	35	741	35	36	3.90	96.10	0.08	99.92
password-crack-c2	898	44026	60	15968	60	71	6.68	93.32	0.16	99.84
password-crack-c3	898	44026	47	1055	47	72	5.23	94.77	0.16	99.84
penetration-c1	375	1807	58	319	58	59	15.47	84.53	3.27	96.73
penetration-c2	375	1807	25	180	25	86	6.67	93.33	4.76	95.24
vpnfilter-c1	678	3076	14	274	14	14	2.06	97.94	0.46	99.54
vpnfilter-c2	678	3076	16	604	16	18	2.36	97.64	0.59	99.41
average	667.62	13632.67	179.19	5509.33	179.19	187.62	27.22	72.78	2.26	97.74

Graph reduction results after causality analysis and edge merge. Average node reduction (by causality analysis) is 72.78%. Average edge reduction (by causality analysis and edge merge) is 97.74%.

TABLE V: Filtering Results

Cases	#Critical Edges	Missing Point	#Non-critical Edges at Missing Point
2File	3	9.49	0
3File	4	6.49	0
Python-wget	4	5.19	1
Python-wget-unzip	8	< 0.01	60
Shell-script	4	3.15	1
Shell-wget	4	0.04	6
Shell-wget-unzip	6	2.83	3
USB-merge	6	2.11	0
curl	4	12.80	1
scp	3	8.29	4
wget	2	6.20	29
command-injection-c1	2	17.00	49
command-injection-c2	3	286.50	0
data-leakage	5	302.89	0
password-crack-c1	2	9.00	34
password-crack-c2	4	14.20	1
password-crack-c3	4	< 0.01	57
penetration-c1	3	0.02	5
penetration-c2	11	< 0.01	21
vpnfilter-c1	2	4.67	12
vpnfilter-c2	3	3.60	15
average	4	32.92	14.24

weights to 1.

- *LocalProjection*: This is the one employed in REPTRACKER (Section IV-D). We locally cluster the incoming edges of every node using Multi-KMeans++ and compute the projection vector using extended LDA.

Evaluation Metric. A better approach will lead to a higher POI reputation in the *HighRP* setting and a lower POI reputation in the *LowRP* setting. We compute the *average percentage improvement* of reputation scores of *GlobalProjection*, *GlobalProjectionNoOutlier*, and *LocalProjection* over *ManualProjection*. Specifically, in the *HighRP* setting, the percentage improvement is calculated as $\frac{RP - RP_{ManualProjection}}{RP_{ManualProjection}}$. In the *LowRP* setting, the percentage improvement is calculated as $\frac{RP_{ManualProjection} - RP}{RP_{ManualProjection}}$.

Evaluation Results. Tables VI and VII show the reputation results of POI nodes for the 21 cases in the *HighRP* setting. Tables VIII and IX show the corresponding results in the *LowRP* setting. For presentation simplicity, we denote the four table categories as *RepCase-HighRP*, *AttCase-HighRP*, *RepCase-LowRP*, and *AttCase-LowRP*.

We have the following observations: (1) *GlobalProjection* performs worse than *ManualProjection* in many cases. The average percentage improvement over *ManualProjection* is negative for *RepCase-HighRP*, *RepCase-LowRP*, and *AttCase-LowRP*, and slightly positive for *AttCase-HighRP*; This shows that dependency graph is quite diverse and it is difficult to separate all edges into two discriminative groups. (2) The performance of *GlobalProjectionNoOutlier* significantly improves over *GlobalProjection*. Compared to *ManualProjection*, *GlobalProjectionNoOutlier* achieves up to 16% average improvement in *HighRP* settings and up to 78% average improvement in *LowRP* settings. This shows the effectiveness and necessity of treating outlier edges differently when doing weight computation; (3) *LocalProjection* achieves the best performance in most of the cases in *HighRP* and *LowRP* settings. Specifically, compared to *ManualProjection*, *LocalProjection* achieves up to 22% average improvement in *HighRP* settings and up to 92% average improvement in *LowRP* settings. The results clearly show the necessity and superiority of clustering and projecting edges locally for each sink node. Note that this approach also treats outliers locally by directly setting their weights to 1, and thus *LocalProjection* embraces the merits of *GlobalProjectionNoOutlier* and achieves better performance. By employing this scheme to compute weights, REPTRACKER effectively identifies benign (*HighRP*) and malicious payloads (*LowRP*) by propagating initial reputation from seeds.

VI. DISCUSSION

A. Alternative Approaches for Weight Computation

As shown in Section V-C, the *Local Clustering and Projection* approach employed in REPTRACKER achieves the best performance in all the compared weight computation approaches, especially much better than empirically setting a fixed projection vector. As mentioned in Section IV-D, supervised learning approaches face serious generalization problems due to the lack of enough training samples and the highly specialized context introduced by the POI event. For unsupervised learning approaches, approaches based on anomaly detection [15] might be a substitution for KMeans. In order to compute the best projection vector, we extend the standard LDA when the within-cluster scatter matrix S_w is singular. We acknowledge that there might be other ways to extend the standard LDA and other methods to achieve discriminative dimensionality reduction [40], [46], which we leave for future exploration.

TABLE VI: Reputation Results Of Representative Cases for Key System Interfaces (seeds RP = 1.0)

Case	ManualProjection	GlobalProjection	Comparison(%)	GlobalProjectionNoOutlier	Comparison(%)	LocalProjection (REPTRACKER)	Comparison(%)
3File	8.51E-01	5.39E-01	-36.64	1.00E+00	17.44	1.00E+00	17.44
2File	8.02E-01	5.27E-01	-34.26	1.00E+00	24.70	1.00E+00	24.70
USB-merge	9.18E-01	9.93E-01	8.16	9.99E-01	8.75	9.79E-01	6.64
shell-script	6.50E-01	7.16E-01	10.18	9.02E-01	38.69	9.46E-01	45.57
curl	9.15E-01	9.21E-01	0.68	9.85E-01	7.65	1.00E+00	9.25
wget	9.82E-01	9.83E-01	0.12	1.00E+00	1.86	1.00E+00	1.85
python-wget	8.05E-01	6.51E-01	-19.09	8.82E-01	9.55	9.99E-01	24.08
python-wget-unzip	6.47E-01	5.63E-01	-12.99	7.16E-01	10.58	7.57E-01	16.88
scp	6.85E-01	8.80E-01	28.33	9.69E-01	41.30	9.58E-01	39.78
shell-wget	8.00E-01	9.10E-01	13.83	7.56E-01	-5.49	9.58E-01	19.84
shell-wget-unzip	6.81E-01	7.45E-01	9.38	8.21E-01	20.59	9.37E-01	37.58
average	7.94E-01	7.66E-01	-2.94	9.12E-01	15.97	9.58E-01	22.15

TABLE VII: Reputation Results Of Key Steps in APT Attacks (seeds RP = 1.0)

Case	ManualProjection	GlobalProjection	Comparison(%)	GlobalProjectionNoOutlier	Comparison(%)	LocalProjection (REPTRACKER)	Comparison(%)
command-injection-c1	8.90E-01	9.65E-01	8.39	9.98E-01	12.07	9.98E-01	12.07
command-injection-c2	7.00E-01	5.00E-01	-28.53	9.69E-01	38.36	9.92E-01	41.64
data-leakage	6.76E-01	7.22E-01	6.82	7.95E-01	17.67	1.00E+00	47.91
password-crack-c1	9.40E-01	9.74E-01	3.63	1.00E+00	6.41	1.00E+00	6.41
password-crack-c2	9.11E-01	9.63E-01	5.72	9.88E-01	8.52	1.00E+00	9.78
password-crack-c3	9.89E-01	8.57E-01	-13.42	1.00E+00	1.06	9.99E-01	1.01
penetration-c1	8.83E-01	9.92E-01	12.31	9.92E-01	12.31	9.67E-01	9.45
penetration-c2	7.95E-01	8.53E-01	7.23	7.88E-01	-0.87	9.47E-01	19.10
vpnfilter-c1	9.55E-01	9.75E-01	2.11	9.92E-01	3.92	9.92E-01	3.92
vpnfilter-c2	9.67E-01	9.99E-01	3.34	9.99E-01	3.36	9.99E-01	3.36
average	8.71E-01	8.80E-01	0.76	9.52E-01	10.28	9.89E-01	15.47

TABLE VIII: Reputation Results Of Representative Cases for Key System Interfaces (seeds RP = 0.0)

Case	ManualProjection	GlobalProjection	Comparison(%)	GlobalProjectionNoOutlier	Comparison(%)	LocalProjection (REPTRACKER)	Comparison(%)
3File	1.49E-01	4.61E-01	-209.97	4.22E-05	99.97	4.22E-05	99.97
2File	1.98E-01	4.73E-01	-138.48	3.07E-04	99.85	3.07E-04	99.85
USB-merge	8.17E-02	6.72E-03	91.77	1.34E-03	98.35	2.07E-02	74.70
shell-script	3.50E-01	2.84E-01	18.92	9.82E-02	71.92	5.35E-02	84.70
curl	8.48E-02	7.86E-02	7.32	1.48E-02	82.50	1.71E-04	99.80
wget	1.83E-02	1.71E-02	6.59	6.16E-05	99.66	1.73E-04	99.06
python-wget	1.95E-01	3.49E-01	-78.68	1.18E-01	39.35	1.48E-03	99.24
python-wget-unzip	3.53E-01	4.37E-01	-23.86	2.84E-01	19.43	2.43E-01	31.01
scp	3.15E-01	1.20E-01	61.75	3.14E-02	90.01	4.19E-02	86.68
shell-wget	2.00E-01	8.97E-02	55.21	2.44E-01	-21.91	4.16E-02	79.25
shell-wget-unzip	3.19E-01	2.55E-01	20.02	1.79E-01	43.95	6.31E-02	80.21
average	2.06E-01	2.34E-01	-17.22	8.83E-02	65.74	4.24E-02	84.95

TABLE IX: Reputation Results Of Key Steps in APT Attacks (seeds RP = 0.0)

Case	ManualProjection	GlobalProjection	Comparison(%)	GlobalProjectionNoOutlier	Comparison(%)	LocalProjection (REPTRACKER)	Comparison(%)
command-injection-c1	1.10E-01	3.52E-02	67.96	2.47E-03	97.75	2.47E-03	97.75
command-injection-c2	3.00E-01	5.00E-01	-66.66	3.11E-02	89.61	8.15E-03	97.28
data-leakage	3.24E-01	2.78E-01	14.23	2.05E-01	36.85	1.82E-04	99.94
password-crack-c1	6.04E-02	2.63E-02	56.43	1.27E-04	99.79	1.27E-04	99.79
password-crack-c2	8.92E-02	3.71E-02	58.39	1.16E-02	86.99	1.29E-04	99.86
password-crack-c3	1.06E-02	1.43E-01	-1,247.74	1.84E-04	98.27	6.02E-04	94.34
penetration-c1	1.17E-01	8.25E-03	92.95	8.25E-03	92.95	3.35E-02	71.38
penetration-c2	2.05E-01	1.47E-01	28.07	2.12E-01	-3.37	5.30E-02	74.14
vpnfilter-c1	4.51E-02	2.49E-02	44.76	7.71E-03	82.91	7.71E-03	82.91
vpnfilter-c2	3.33E-02	1.00E-03	96.99	7.97E-04	97.61	7.97E-04	97.61
average	1.29E-01	1.20E-01	-85.46	4.79E-02	77.94	1.07E-02	91.50

B. Parallelization

Part of the construction of weighted dependency graphs can be potentially parallelized with distributed computing. The dependency graph produced by traditional causality analysis [29], [30] can be parallelized by searching dependency separately. The feature generation for each edge is independent and is a good candidate for parallelization. In the scenarios where multiple hosts are involved, dependencies on each host can also be pre-computed in parallel and cross-host causality analysis thus becomes the concatenation of multiple generated dependency graphs. Nonetheless, the weight computation and the reputation propagation cannot be easily separated due to dependencies on the computation results of a set of nodes, and the massive dependencies among system events also pose significant challenges to parallel processing. Thus, weighted causality analysis with parallelization is an interesting research direction that requires non-trivial efforts.

C. Attacks Against the Proposed Framework

In practice, the attacker, with some knowledge about the proposed system, may optimize its attack to stay under cover. For instance, (i) to have a low time weight, the attacker may inject its malicious file earlier but start its attack later, or (ii) to have lower data weight, the attacker may attack using multiple processes each with small data amounts (*i.e.,* distribute the malicious behavior to multiple processes). An attacker may also choose to gain reputation first before attacking the system or stay under cover and attack probabilistically. In this paper, we do not consider the potential attacks against the reputation system and we leave them for future work.

D. Industrial View

Because APT attacks consist of many small steps over a long period of time, even though security experts can get the system wide log, it is time-consuming to track and reconstruct the attack and hard to discover the complete attack steps. Moreover, depends on the individuals capability, the quality of the analysis may vary. REPTRACKER not only reduces the time consumption of the analysis but also removes a dependency on the capability of the security experts. Hence, by applying REPTRACKER, we can guarantee a certain level of analysis result automatically. The automated process is vital to keep the security level as high as possible. Since it is possible to reconstruct the attack steps in an individual system automatically, it is highly likely to be applicable to a small scale business that is not affordable to get help from security experts when security incident happens.

RepTracker can be useful not only for post-mortem analysis but also for capturing symptoms before the security incident. According to the recent malware report [32], fileless malware such as PowerShell attack is surging 432% in 2017. The fileless malware does not leave any files in the system so the anti-malware solution is experiencing difficulties. System-wide logging can capture the activity of the process by observing the reputation of connected IP address periodically, RepTracker can give an early warning to the user before or during the attack on the fly.

VII. Related Work

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

Causality Analysis. Causality analysis based on system monitoring data plays a critical role in security analysis. King et al. [29] proposed a backward causality technique to perform intrusion analysis by automatically reconstructing a series of events that are dependent on a user-specified event. It further improves the approach to track dependency cross different hosts [30]. Goel et al. [24] proposed a technique that recovers from an intrusion based on forensic analysis.

Further efforts have been made to mitigate the dependency explosion problem by performing fine-grained causality analysis. BEEP [34] identified the event handling loops in programs and enable selective logging for unit boundaries and unit dependencies through instrumentation. Ma et al. [38] alternates between logging and taint tracking to derive more accurate dependencies, and Kwon et al. [31] further leverages context-sensitive causal models to improve the analysis. PrioTracker [36] proposed to learn a reference model of normal activities to prioritize abnormal activities.

Compared to these existing works, RepTracker addresses the dependency explosion problem via computing weights for each dependency based on our novel features and pruning edges with low weights, which do not require heuristics [29], [30], binary instrumentation [31], [38], modification of kernels [11], or reference models of normal activities [36]. Moreover, RepTracker provides reputation propagation that can automatically identify malicous payloads, which cannot be done by all these works.

To mitigate the burden of storing large amount of system monitoring data, LogGC [35] leveraged garbage collection to remove temporary files, Xu et al. [51] proposed to merge dependencies while still preserving high-fidelity causal dependencies, and Tang et al. [49] used templates to summarize the set of system libraries loaded at the process initiation period. RepTracker can be integrated with these works to achieve better scalability for handling cases that span a long period of time (*e.g.,* months).

Weight Computation. Several components of RepTracker are built up on a set of existing techniques. Our local edge clustering step is based on KMeans++ [10], which optimizes the seed initialization and leads better clustering quality, compared with KMeans. Our local feature projection step is constructed based on linear discriminant analysis (LDA) [40], which finds a linear combination of features that characterizes or separates multiple classes of objects. Yet, to apply it in our setting, we extend the standard LDA to handle the challenges including lack of a prior class information, matrix singularity, and limited number of labeled instances.

Reputation Propagation. Our reputation propagation model is inspired by the TrustRank algorithm [26], which is 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 [43], and identifies spam pages as those with low scores. Similar ideas have been applied in applications including Sybil detection on social networks [14], [21], [25]. The model implemented in this work differs in that it propagates the reputation score in an aggregation style rather than a distribution style, which avoids the serious degradation of reputation scores after propagating on long dependency paths.

VIII. Conclusion

We have presented a novel causality analysis approach, RepTracker, which builds a weighted dependency graph from system auditing events that record the information of system calls. RepTracker is designed to address the two major problems faced by the causality analysis: dependency explosion and non-trivial efforts in graph inspection. RepTracker generates three novel types of discriminative features (*i.e.,* relative time difference, relative data amount differences, and concentration degree) for the dependencies among system auditing events, and leverage these features to compute weights for each dependency. Based on these weights, RepTracker can surface attack provenance by hiding low weighted edges. RepTracker further propagates reputation from the seed nodes in the weighted dependency graph to automatically determine whether the reputation of a target file is similar to untrusted sources, saving the efforts in graph inspection. The evaluations on representative cases for key system interfaces and steps in real attacks demonstrate the effectiveness of RepTracker in addressing dependency explosion and graph inspection.

References

[1] "Cve-2014-6271: bash: specially-crafted environment variables can be used to inject shell commands." https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6271.

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

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

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

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

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

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

[8] A. Albert, *Regression and the Moore-Penrose pseudoinverse*. Elsevier, 1972.

[9] E. Alpaydin, *Introduction to machine learning*. MIT press, 2009.

[10] 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.

[11] A. M. Bates, D. Tian, K. R. B. Butler, and T. Moyer, "Trustworthy whole-system provenance for the linux kernel," in *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, 2015, pp. 319–334.

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

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

[14] Q. Cao, M. Sirivianos, X. Yang, and T. Pregueiro, "Aiding the detection of fake accounts in large scale social online services," in *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX, 2012, pp. 197–210. [Online]. Available: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/cao

[15] 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

[16] cnn, "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.

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

[18] Fireeye, "Anatomy of advanced persistent threats," 2017.

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

[20] 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.

[21] 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 *2018 IEEE Conference on Communications and Network Security (CNS)*, May 2018, pp. 1–9.

[22] 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 *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018, pp. 639–656. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/gao-peng

[23] P. Gao, X. Xiao, Z. Li, F. Xu, S. R. Kulkarni, and P. Mittal, "AIQL: Enabling efficient attack investigation from system monitoring data," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, 2018, pp. 113–126. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/gao

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

[25] N. Z. Gong and D. Wang, "On the security of trustee-based social authentications," *Trans. Info. For. Sec.*, vol. 9, no. 8, pp. 1251–1263, 2014.

[26] 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.

[27] 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.

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

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

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

[31] Y. Kwon, F. Wang, W. Wang, K. H. Lee, W. Lee, S. Ma, X. Zhang, D. Xu, S. Jha, G. F. Ciocarlie, A. Gehani, and V. Yegneswaran, "MCI : Modeling-based causality inference in audit logging for attack investigation," in *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, 2018.

[32] M. Labs, "Threats report 2018," 2018, https://www.mcafee.com/enterprise/en-us/assets/infographics/infographic-threats-report-mar-2018.pdf.

[33] K. H. Lee, X. Zhang, and D. Xu, "High accuracy attack provenance via binary-based execution partition," in *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*, 2013.

[34] ——, "High accuracy attack provenance via binary-based execution partition," in *NDSS*, 2013.

[35] ——, "Loggc: garbage collecting audit log," in *CCS*, 2013.

[36] 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 *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, 2018.

[37] 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 *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018.*, 2018, pp. 241–254.

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

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

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

[41] C. Miller and C. Valasek, "Remote Exploitation of an Unaltered Passenger Vehicle," *Black Hat USA*, vol. 2015, p. 91, 2015.

[42] NPR, "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.

[43] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank Citation Ranking: Bringing Order to the Web," Technical Report 1999-66, 1999.

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

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

[46] M. Sugiyama, "Local fisher discriminant analysis for supervised dimensionality reduction," in *Proceedings of the 23rd international conference on Machine learning*. ACM, 2006, pp. 905–912.

[47] Symantec, "Advanced Persistent Threats: How They Work," 2017, https://www.symantec.com/theme.jsp?themeid=apt-infographic-1.

[48] Sysdig, "Sysdig," 2017, http://www.sysdig.org/.

[49] 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 *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, 2018, pp. 1324–1337.

[50] 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.

[51] 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, pp. 504–516.