Towards a Timely Causality Analysis for Enterprise Security

Yushan Liu*§, Mu Zhang†§¶, Ding Li‡¶, Kangkook Jee‡, Zhichun Li‡¶, Zhenyu Wu‡, Junghwan Rhee‡, Prateek Mittal*¶
*Princeton University, †Cornell University, ‡NEC Labs America
*{yushan,pmittal}@princeton.edu, †mz496@cornell.edu, ‡ {dingli,kjee,zhichun,adamwu,rhee}@nec-labs.com

***Abstract*—The increasingly sophisticated Advanced Persistent Threat (APT) attacks have become a serious challenge for enterprise IT security. *Attack causality analysis*, which tracks multi-hop causal relationships between files and processes to diagnose attack provenances and consequences, is the first step towards understanding APT attacks and taking appropriate responses. Since attack causality analysis is a time-critical mission, it is essential to design causality tracking systems that extract useful attack information in a timely manner. However, prior work is limited in serving this need. Existing approaches have largely focused on pruning causal dependencies totally irrelevant to the attack, but fail to differentiate and prioritize abnormal events from numerous *relevant, yet benign and complicated* system operations, resulting in long investigation time and slow responses.**

To address this problem, we propose PRIOTRACKER, a backward and forward causality tracker that automatically prioritizes the investigation of abnormal causal dependencies in the tracking process. Specifically, to assess the priority of a system event, we consider its rareness and topological features in the causality graph. To distinguish unusual operations from normal system events, we quantify the rareness of each event by developing a *reference model* which records common routine activities in corporate computer systems. We implement PRIOTRACKER, in 20K lines of Java code, and a reference model builder in 10K lines of Java code. We evaluate our tool by deploying both systems in a real enterprise IT environment, where we collect 1TB of 2.5 billion OS events from 150 machines in one week. Experimental results show that PRIOTRACKER can capture attack traces that are missed by existing trackers and reduce the analysis time by up to *two orders of magnitude*.

I. INTRODUCTION

The increasingly sophisticated Advanced Persistent Threat (APT) attacks have become a serious challenge for enterprise IT security. In the past decade, over 6000 severe incidents [1] have been reported. Particularly, large enterprises, such as Target [2] and HomeDepot [3], have been intentionally targeted and suffered significant financial loss and reputational damage. APT attacks are conducted in multiple stages, including initial compromise, internal reconnaissance, lateral movement and eventually mission completion.

An intrusion may be detected at any of the stages. However, detection by itself only reveals unconnected attack traces. Besides, a large portion of individual attack footprints are seemingly insignificant and thus not suspicious enough to raise alarms. Hence, to see the forest from the trees, system administrators must carefully perform *attack causality analysis* [4]–[11], in order to achieve a complete and sound understanding of a detected attack. This is the very first step towards a safe system recovery from cyber attacks. To do so, administrators need to first discover how the adversary gained access to the system, and then determine both exposed and hidden damage inflicted on the system, such as information leakage, compromised files and installed backdoors. More concretely, to identify the sequence of steps in an APT attack, prior work reconstructs multi-hop causal dependencies between OS-level system objects including processes, files and sockets. Starting from a detected event, system dependencies are traced backward and forward in temporal order, so as to eventually reveal the attack provenances and uncover all the consequences, respectively.

Despite the fact that *attack causality analysis* is performed in a post-mortem fashion, it is a considerably time-sensitive mission due to two reasons. First, a compromised system requires complete cleanup before returning to normal operation. Before recovery, financial loss caused by decreased system uptime can easily grow to millions of dollars [12]. A timely causality analysis can accelerate the discovery of all attack traces and reduce such recovery cost. Second, APT attacks are performed in multiple stages. A detected point may not be the very end of attack sequence and the intrusion could further develop to cause more damage. A timely attack causality analysis can help understand attack intentions and prevent future damage.

As a result, we believe a practical security causality analysis must take time limit into account and extract useful attack information in a timely manner. Unfortunately, this has been largely overlooked by the prior work. Previous efforts have mainly focused on addressing the dependency explosion problem [5], [6], [10], [11] via data reduction. Particularly, they have attempted to eliminate *irrelevant* system dependencies via either 1) heuristics-based pruning [4], [5] or 2) binary instrumentation [6], [7] and taint analysis [8], [9]. However, reducing the data volume of *irrelevant* dependencies does not

§This work was conducted when Yushan Liu was an intern at NEC Labs, mentored by Mu Zhang, who was a Researcher at NEC.
¶ Mu Zhang, Ding Li, Zhichun Li and Prateek Mittal are corresponding authors.

Network and Distributed Systems Security (NDSS) Symposium 2018
18-21 February 2018, San Diego, CA, USA
ISBN 1-1891562-49-5
http://dx.doi.org/10.14722/ndss.2018.23254
www.ndss-symposium.org

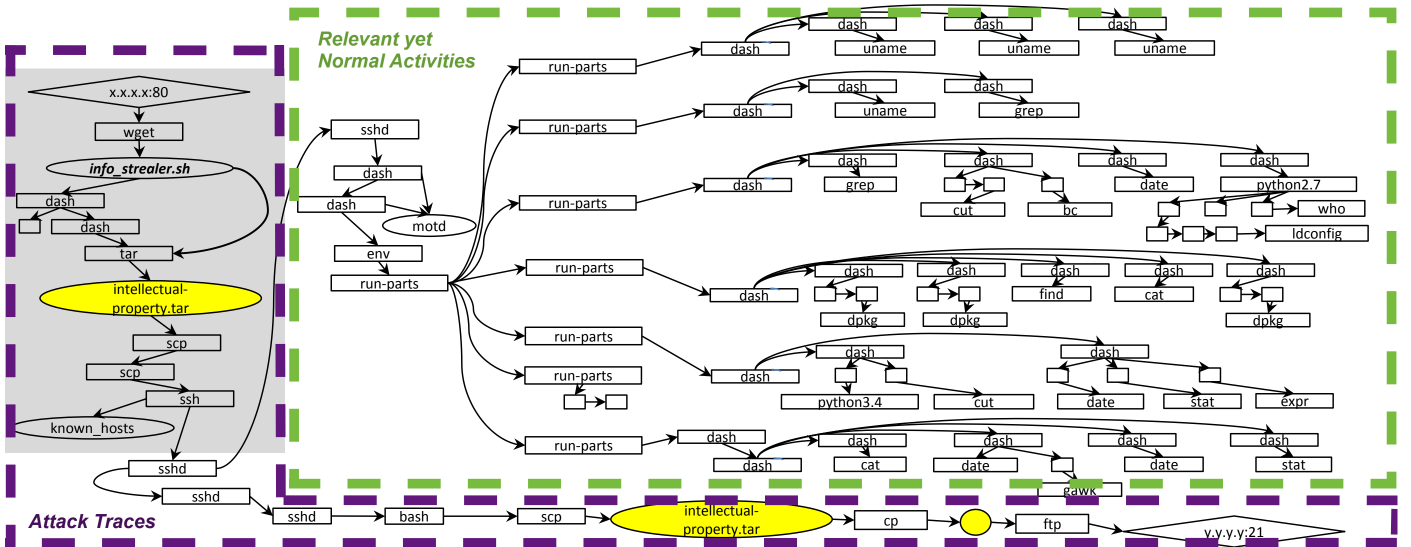

Fig. 1: Reduced Version of Forward Tracking Graph for the Attack Scenario. Rectangles represent processes; ovals denote files; diamonds indicate sockets. Grey background represents one host, while white background denotes the other.

necessarily lead to a decrease in attack investigation time. Prior studies still have to invest excessive time in analyzing *relevant, yet benign and complex* OS events, which dominate the system (see Figure 1 and Section II for an example). Essentially, this is because they lack the capability of differentiating unusual activities from common system operations. Therefore, they have to treat all relevant dependencies, abnormal and normal, equally and simply keep track of every causal relation.

To address this problem, we propose PRIOTRACKER, a backward and forward causality tracker that automatically prioritizes the search for abnormal causal dependencies in the tracking process. We further formalize a time-constrained causality analysis to be an optimization problem, which aims to reveal the maximum number of anomalies within a certain time limit.

To distinguish abnormal operations from normal system events, we quantify the *rareness* of each event by developing a *reference model* which records common routine activities in corporate computer systems. To build such a model, we take full advantage of the homogeneous IT environment in enterprises, and collect normal OS events from copious amounts of peer systems. Consequently, we enable a "crowd-sourcing" based method to distill outliers from regular behaviors.

We associate every event with a *priority score* and select the event with the highest priority score in the process of tracking. The priority score of an event is computed based on its rareness and other topological features in the causality graph. We assign weights to these features, which are optimized using the *Hill Climbing* algorithm to find the maximum number of rare events before a given deadline. Note that although rareness and other topological features are heuristically chosen, their weights are formally assigned using machine learning algorithm to reflect their effectiveness.

Priority-based methods have been widely used in security analyses. Previous efforts have been made to expedite static data-flow analysis [13], symbolic execution [14]–[16], fuzzing [17] and digital forensics [18], [19] through measuring the priority of either program-level constructs or user-level physical entities. In contrast, we enable a priority-based analysis in a completely different domain, and therefore have to address the unique challenge of quantifying priority in OS-level dependency tracking. To the best of our knowledge, we are the *first to accelerate attack causality analysis via identifying and prioritizing abnormal causal relations*.

We implement PRIOTRACKER in 20K lines of Java code, and a reference model builder in 10K lines of Java code. Our experiments are performed on 54 Linux and 96 Windows machines used daily by researchers, developers and administrators in an anonymous IT enterprise. Over ten months, we use an audit log system to capture OS-level events from host machines and store them in a database. We also record the common system operations to build the reference model. We evaluate our tool on 8 attack cases, which involve 2.5 billion OS events spanning one week, and 75 points of interest, which generate 429,900 sophisticated causal relations. Experimental results show that PRIOTRACKER can capture attack traces that are missed by existing trackers and can reduce the analysis time by up to *two orders of magnitude*.

In summary, this paper makes the following contributions:

- We are the first to formalize timely attack causality analysis and to introduce priority to *attack graph construction*. We present PRIOTRACKER, an anomaly-prioritized backward and forward causality tracker which computes the priority score of a causal dependency based on its rareness and topological characteristics in the causality graph. We leverage the Hill Climbing algorithm to optimize the feature weights in the priority score.
- We create a reference model via observing the OS-level activities from peer systems in homogeneous enterprise hosts. Based upon this model, we propose

a "crowd-sourcing" based method to differentiate unusual behaviors from normal ones, and thus to assist the computation of the priority score. Our reference model is able to be customized based upon the system events collected from any specific enterprise IT environment.

- We have implemented PRIOTRACKER and a reference model builder, and deployed them into a real-world enterprise computer environment. We collect a dataset that is orders of magnitude larger than the ones used in previous work [9], [20]. Our experimental results are promising, showing that PRIOTRACKER can find attack related activities significantly faster than the state-of-the-art technique.

II. OVERVIEW

In this section, we explain the notion of causality analysis and forward tracking graph via a motivating attack scenario example. Next, we introduce the problem statement, system architecture and threat model.

A. Motivating Example: Forward Tracking the Impact of Insider Related Data Leaks

1) **Attack Scenario:** An employee worked at a computer networking company which services a customer in the semiconductor industry. In order to do business with the semiconductor firm, the networking company had access to the customer's critical server which stored its most sensitive intellectual property. When the networking company employee got his new job in another semiconductor firm, he used his remaining time at his old job to steal the sensitive data. To do so, he downloaded a malicious BASH script to the data server via *HTTP*, and executed the script in order to discover and collect all the confidential documents on the server. Then, he compressed the files into a single tarball, transferred the tarball to a low-profile desktop computer via *SSH*, and finally uploaded it to the file server via *FTP* under his control. Note that similar attack scenarios have happened in the real-life insider incidents of DuPont [21], Barclays [22], Ellery Systems [23], etc.

2) **Causality Analysis:** The incident was eventually caught manually by his colleagues in the new company, and thus reported to the victim semiconductor firm. The corporate IT administrators then started an investigation and discovered the malicious script on the data server. Furthermore, to fully recover from this attack, they also expected to locate and destroy all the copies of leaked sensitive files, so that these copies would not be accessed by any other unauthorized personnel in the future. To this end, they leveraged attack causality analysis [4], [5] to conduct causal dependency forward tracking, which connects the OS-level objects (files, processes and sockets) via system events in temporal order.

3) **Forward Tracking Graph:** Figure 1 demonstrates the resulting dependency graph of forward tracking in this attack case. In the dependency graph, each node represents a process, file or network socket. An edge between two nodes indicates a system event involving two objects (such as process creation, file read or write, network access, etc.). Multiple edges are chained together based on their temporal order.

Particularly, Figure 1 exposes all the subsequent system events that are caused by the data exfiltration incident. The graph begins with the network event where malicious script *info_stealer.sh* is downloaded by *wget* from x.x.x.x:80 to the server machine. The script is then executed in *dash*, which consequently locates sensitive files and triggers *tar* to compress the discovered documents into one single file, *intellectual-property.tar*. The tarball is further delivered to another Linux desktop using the $scp \leadsto ssh \leadsto sshd \leadsto scp$ channel. Once the file has reached the desktop system, a new copy is made and eventually sent to remote cite y.y.y.y:21 through *ftp*.

In the meantime, the result graph also reveals that *sshd* executes massive Linux commands through triggering a series of *run-parts* programs. In fact, many of these Linux commands are intended to update the environmental variables, such as *motd* (i.e., message of the day), so as to create a custom login interface. These are *relevant* activities that are caused by *scp* operation but are relatively more common behaviors compared to transferring a previously unseen file. However, existing causality trackers cannot differentiate them from the real attack activities. Thus, they may spend a huge amount of time analyzing all the events introduced due to *run-parts*, even before studying data breach through *ftp*. To our experience, this could delay the critical attack investigation for a significant long period of time, ranging from minutes to hours depending on different cases. Unfortunately, Verizon Data Breach Report [24] discovered that nearly 90 percent of intrusions saw data exfiltration just minutes after compromise. Thus, any delay in incident response literally means more lost records, revenue and company reputation.

In this case, the large causal graph is caused mostly by intensive process creations. Process forking leads to a greater amount of dependencies particularly in forward tracking than in backtracking because one process only has one parent but may have multiple children. However, it is noteworthy that, the delay of attack inspection is a common problem for both forward and backward dependency tracking. Excessive file or network accesses can also take up a significant portion of analysis time in both practices.

Also note that, the lack of analysis priority is orthogonal to the data quantity problem which has been intensively studied by prior data reduction efforts [6], [7], [25]. Even if the overall data volume has been reduced, a security dependency analysis, without distinguishing between common and uncommon actions, can still be much delayed due to tracking the huge amount of normal activities.

B. Problem Statement

To address this problem, we propose PRIOTRACKER, which prioritizes the investigation of abnormal operations based upon the differentiation between routine and unusual events. Concretely speaking, we expect PRIOTRACKER to meet the following requirements.

- **Accuracy.** Given sufficient analysis time, our causality tracker must capture all the critical activities. It must not miss system events caused by attacks.
- **Time Effectiveness.** Incident response is time critical and thus a practical attack investigation must

be subject to time constraints. Given limited analysis time, our dependency tracking system must find the maximum number of highly abnormal behaviors.

- **Runtime Efficiency.** The proposed prioritization technique must not introduce a significant amount of additional runtime overhead to the underlying dependency tracking system.

Particularly, when analyzing the aforementioned attack scenario, we hope PRIOTRACKER to directly reach the *ftp* branch without touching the majority of *run-parts* branch in advance, so that provided a temporal limit is applied to the analysis, the real attack can still be revealed in time.

1) **System Architecture.:** To achieve these goals, we design the architecture of our system, depicted in Figure 2. PRIOTRACKER consists of three major components, i.e., a priority-based causality tracker, a reference model builder and a reference database. Our system is designed to be deployed in a large-scale and homogeneous enterprise IT environment. In this environment, OS-level events are collected from every individual host and are pushed to a stream processing platform, and are eventually stored into the event database.

We retrieve low-level system events from Linux and Windows machines using kernel audit [26] and ETW kernel event tracing [27], respectively. Specifically, we collect three types of events: 1) file events, including file read, write and execute, 2) process events, such as process create and destroy, and 3) network events, including socket create, destroy, read and write.

Our reference model builder subscribes to the stream in order to count the occurrences of the same events over all the hosts. The computed occurrences are then saved into our key-value store -based reference database so that they can be efficiently queried by causality tracker. Once an incident happens, the triggering event is presented to our causality tracker to start a dependency analysis. The causality tracker will consequently search for related events from database. At the same time, it also queries reference database in order to compute the priority score for the events to be investigated. An event bearing higher priority score will be analyzed first. In the end, the causal dependencies are generated based upon event relationships, and are presented as result graphs for further human inspection.

2) **Threat Model.:** We follow the threat model of previous work [4]–[9], [28]. Particularly, we define the trusted computing base (TCB) for causality analysis to be the kernel mechanisms, the backend database that stores and manages audit logs, and the causality tracker. With respect to our TCB, we assume that audit logs collected from kernel space [26], [27] are not tampered, since kernel is trusted. Kernel-level attacks that deliberately compromise security auditing systems are beyond the scope of this study.

We do consider that external attackers or insiders have full knowledge of "normal" activities, so that they can intentionally craft attacks with seemingly normal operations and may poison our reference database using a burst of repeated malicious activities.

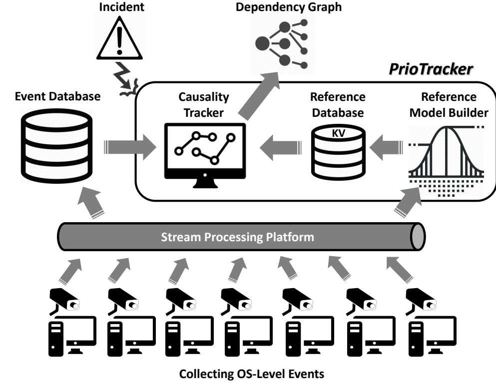

Fig. 2: Architecture Overview of PRIOTRACKER.

III. TIME-CONSTRAINED ANOMALY PRIORITIZED CAUSALITY TRACKING

In this section, we present the design details of time-constrained anomaly prioritized causality tracking. First, we give the basic algorithm of PRIOTRACKER. Next, we discuss the features considered when computing the priority score of a system event. Then, we introduce the Hill Climber algorithm used for weight assignment in the priority score.

A. Basic Algorithm

In practice, attack investigation time is not unlimited. Our PRIOTRACKER considers time as a key factor and aims to track more abnormal behaviors with higher potential impact with a certain time limit.

Tracking tasks start from a detection point, which usually is an intrusion alert detected by the monitoring system. Algorithm 1 illustrates our basic algorithm to perform a time-constrained causality tracking. In general, we follow the prior technique [4] to build the dependencies between OS-level events. However, to enable timely security causality analysis, we prioritize the dependency tracking of abnormal events, in contrast to the previous work which blindly selects the next event for processing.

More concretely, our dependency tracker internally maintains a priority queue PQ to hold all the events that wait for processing. This queue is sorted in descending order based on the priority scores of enclosed events, so that the event with highest priority is always placed at the head and will be processed first. Upon receiving a starting event se, our tracker computes its priority score using function $Priority()$ and adds it into this queue. Then, PRIOTRACKER iteratively processes each item until the queue becomes empty or the given analysis time limit T_{limit} is reached. In each iteration, it fetches an event from the head of queue, adds this event to the result graph G, and invokes COMPUTEDEPS() to compute its causal dependencies based on temporal relationships [4]. COMPUTEDEPS() returns a set of events E for further analysis. Then, we compute the priority score for each element in this set before inserting them into the priority queue. In the end, Algorithm 1 outputs the dependency graph G for forensic analysis. Events that are not tracked within the time limit

Algorithm 1 Dependency Tracking Algorithm

```
1: procedure PRIOTRACK(se, T_limit)
2:     PQ ← ∅
3:     PQ.INSERT(se, Priority(se))
4:     while !PQ.ISEMPTY() and T_analysis < T_limit do
5:         e ← PQ.DEQUEUE()
6:         G ← G ∪ e
7:         E ← COMPUTEDEPS(e)
8:         for ∀e' ∈ E do
9:             PQ.INSERT(e', Priority(e'))
10:        end for
11:    end while
12:    return G
13: end procedure
```

are not included in the resulting graph but are stored in the database for further analysis.

PRIOTRACKER supports across-host tracking by performing IP channel event matching. For an IP channel event on host A talking to host B, we search for its match on host B with the reverse of the IP and port information, which are, within some tolerance, occurring at the same time.

B. Priority Score

1) **Important Factors***:* We consider three factors to be important when determining the priority of system events to be processed.

- **Rareness of Events.** In general, attack behaviors and malware activities are deviated from massive normal operations. Particularly, APT incidents often enable zero-day attacks, which by nature have never been observed in regular systems. As a result, special attention needs to be paid to rarer events compared to routine activities.
- **Fanout.** As illustrated in our motivating example, routine system operations can be performed in a batch, which consists of multiple sub-operations. Besides, regular system activities (e.g., creating or accessing numerous temporary files) may happen periodically over time. This in turn generates events with very high fanout in a dependency graph (up to tens of thousands), which does not contribute to attack forensics. In addition, analysis of causalities with high fanout can be very time-consuming and therefore may delay or even disable timely investigation of other attack traces. Essentially, there exists a trade-off between time effectiveness and analysis coverage, where a balance needs to be struck.
- **Dataflow Termination.** To invade an enterprise system, attackers have to first exert an external influence on internal system objects (e.g., malware dropping, malicious input to vulnerable network services, etc.) to persist; then, they can further use the compromised persistent objects (e.g., malicious executables, victim long-running services) to cause impact on other parts of the system. Consequently, a file without being written in the past is less critical for backtracking intrusions; a file the has never been read or executed so far is less interesting for tracking attack consequences forward. The former one is introduced by prior work as the "read-only" pruning heuristic [4] in backtracker. The latter case, however, cannot be completely ignored because a currently "write-only" file may still be accessed at a future point.

Hence, to generate the priority score for each event, we need to first compute the scores for edge rareness, fanout and dataflow termination, respectively.

2) **Rareness Score***:* First, we define the rareness score of an event $rs(e)$ base upon our reference model:

$$rs(e) = \begin{cases} 1, & \text{if } e \text{ has not been observed by reference model} \\ \frac{1}{ref(e)}, & \text{otherwise} \end{cases}$$

$ref(e)$ is the reference score of event e, which is computed by reference model according to the historical occurrence of e. We elaborate the computation of reference score in Section IV.

3) **Fanout Score***:* Second, we formalize the fanout score of an event $fs(e)$ to be the reciprocal of its fanout: $fs(e) = \frac{1}{fanout(e)}$. An event with higher fanout score will be examined first. Note that when we compute fanout, we do not consider outgoing socket edges whose destinations are external networks or specific internal servers (e.g., DNS), which are not under our monitoring and thus will not be further tracked in the first place.

We prefer edges with low fanout due to the consideration of both security and efficiency. Analyzing causal relations with huge fanout is often very slow because dependencies grow exponentially. Thus, putting them first may lose the chance to explore other system dependencies which could also be caused by attacks. In contrast, analysis of causalities with lower fanout is comparatively simpler and costs much less time to complete. Even if, in the worst-case scenario, fast-tracking an event with low fanout does not reveal any attack traces, it only introduce a small amount of delay to the examination of other complex causalities.

We admit, as a potential evasion technique, an attacker may attempt to leverage system causality with high fanout to hide their attack footprints, in order to delay our analysis. However, it is worth noting that, though we deprioritize paths with high fanout, we do not prune off them as prior work [4] does. If an attack is indeed buried in branches bearing high fanout, given enough time and computation resources, our tracker can eventually reach that point. Besides, an attack cannot be launched solely using complex dependencies with high fanout, while the other portion of attack-related causalities can still be discovered by our approach from numerous normal edges in a *faster* fashion. Since the entire attack footprints are logically connected, any uncovered portion can help human experts find the remaining ones. On the contrary, without prioritization, processing benign dependencies with huge fanout can excessively consume computing resources. Consequently, none of attack traces can be reached before analysis deadline, and therefore the entire attack is missed.

4) **Dataflow Termination***:* Terminated dataflow is a special case, where fanout equals zero. Therefore, we complete

our definition of *fanout score* by also checking whether an event has further impacts:

$$fs(e) = \begin{cases} 0, & \text{if } e \text{ reaches a read-only file in backtracking} \\ \sigma, & \text{if } e \text{ reaches a write-only file in forward tracking} \\ \frac{1}{fanout(e)}, & \text{otherwise} \end{cases}$$

Hence, if backward dataflow is terminated due to read-only files, we deprioritize the analyses of associated events via assigning 0 to the score. However, when forward dataflow ends with "write-only" files, we do not completely rule out the possibility that these files will later be accessed. Therefore, we instead give them a lower but non-zero score σ. Empirically, we set σ to be 0.3.

5) **Priority Score***:* The priority score of each event can be derived from the composition of these factors.

Definition 1. The *Priority Score* of a system event, $Priority(e)$, is the weighted sum of rareness score $rs(e)$ and fanout score $fs(e)$:

$$Priority(e) = \alpha \times rs(e) + \beta \times fs(e) \tag{1}$$

, where α and β are the weights that need to be determined. An event with higher priority score will be investigated first.

C. Weight Assignment

The next step is to give a proper weight to each parameter of the priority function. Ideally, when weights are correctly assigned, we expect our dependency tracker to find the maximum amount of *attack traces* within a finite time bound. Nevertheless, it is very hard, if not impossible, to measure the relatedness between a single event between two OS-level objects and an attack, especially before the attack is completely known. This is by nature due to the diversity and randomness of cyber crimes committed by human attackers, and by itself can be a challenging research problem. Therefore, to date, expert knowledge has to be kept in the loop to evaluate automatically generated security causality graphs and to draw a decisive conclusion.

To address this problem, we instead use *rareness* as a metric to approximate the connection between a causal relation and unknown attacks. As a result, our goal of weight assignment is to enable our tracker to uncover as many unusual events as possible within a certain time limit. Admittedly, an adversarial could utilize many normal system operations when launching an attack, and therefore the overall amount of rare events does not necessarily indicate the presence of attacks. However, at certain points of a stealthy crime, an attacker has to perform some harmful and thus abnormal operations, such as data exfiltration or system tampering, in order to serve the purpose. Then, a discovery of more unusual activities may increase the chance of capturing real attack footprints.

To achieve the discovery of the maximum number of unusual events, we need to strike a balance among the aforementioned factors. On one hand, at every step of dependency tracking, we always expect to choose a rare and impactful event over a common or uninteresting one. On the other hand, we also hope to quickly explore the entire search space, and find the direction that leads to more rare activities. Essentially, this is a global optimization problem, which we define as follows:

Definition 2. The *Weight Assignment* is an optimization problem to maximize the result of an objective function for a given set of starting events E:

$$\begin{aligned} \max \quad & f(E,(\alpha,\beta)) = \\ & \sum_{e \in E} EdgeCount_\theta(PrioTrack_{(\alpha,\beta)}(e, T_{limit})) \\ s.t. \quad & 0 \le \alpha \le 1, \alpha + \beta = 1 \end{aligned} \tag{2}$$

, where α and β are the weight parameters for rareness, fanout and dataflow scores respectively. These scores are further used to derive the priority score in dependency tracking $PrioTrack$. $EdgeCount$ function counts the number graph edges whose rareness score is greater than a given threshold θ. Empirically, we set θ to be 0.1 and set time limit T_{limit} to be 60 minutes. Note that these values can be customized for specific environments and security requirements.

We then utilize the *Hill Climbing* [29] algorithm to achieve the optimization of Equation 3. This algorithm can gradually improve the quality of weight selection via feedback based method. We have implemented such a feedback loop, which takes a set of starting events E and an initial weight vector (α, β) as inputs. To create the starting event set E, we randomly select 1,113 system events, within a timespan of 10 months from August 2016 to May 2017, which lead to excessively large dependency graphs (up to 73,221 edges with 2,391 edges on average). At each iteration, the algorithm adjusts an individual element in the weight vector and determines whether the change improves the value of objective function $f(E,(\alpha,\beta))$. If so, such a positive change is accepted, and the process continues until no positive change can be found any more. Eventually, the algorithm produces the optimized weight parameters, where $\alpha = 0.27$ and $\beta = 0.73$.

Note that the rareness and fanout features demonstrate a trade-off between analysis coverage and time effectiveness. The fact that the weight of fanout is three times as much as that of rareness indicates the trained tracking system prefers to quickly expand the search area to reach a global optimal. As a result, on one hand, it tends to prioritize low-fanout events and avoid high-fanout events that cause the search to sink into a very busy local neighborhood. On the other hand, it depends less on the rareness score of the current event under examination because it cannot adequately reflect the overall rareness of following events. This in fact reveals a limitation of our reference model, which quantifies rareness in a context-insensitive fashion. We discuss the potential improvement in Section VI.

D. Implementation

We have developed the priority-based dependency tracker in 20K lines of Java code. When acquiring the enabling information (i.e., rareness, fanout and write-only/read-only), we pay special attention to runtime efficiency in order to cope with the massive amounts of system events collected from large enterprises. Particularly, we introduce several optimization techniques to accelerate data query.

1) **In-Memory Key-Value Store***:* Our tracking algorithm requires frequent access to reference database in order to query reference score of individual events. Traditional database persisted on hard disks cannot satisfy such performance

requirements. As a result, we store the reference data in RocksDB [30], which on one hand enables an in-memory key-value store for fast access, and on the other hand can still persist data in the traditional way.

2) **Event Cache***:* To compute the fanout of an event or to determine if an event reaches a read-only or write-only file, we enable a look-ahead method to examine a further one hop of dependencies. In fact, these additional query results are not only used for the current computation of priority scores, but also later become part of result dependency graph. Thus, to avoid redundant query overhead, we cache these results for future usages.

3) **Look-Ahead with a Limit***:* Sometimes, the fanout of an event is extremely high. For instance, a Firefox process may touch hundreds of temporary files. In this case, counting the exact fanout via database query is very time-consuming, and could lead to degradation of runtime efficiency. Besides, in such a case, the exact value of fanout becomes less interesting in terms of computing and comparing the priority score. Therefore, we approximate the fanout by putting a limit n on the query, so that it only looks for the first n events that are dependent on the current one. In effect, if the fanout is greater than n, the fanout score $fs(e)$ is in practice defined to be $1/n$ instead of $1/fanout(e)$.

IV. Reference Model

In this section, we elaborate on the reference model, which quantifies the rareness of system events and helps distinguish the anomalies from noisy normal system operations. First, we give the details of data collection in an enterprise IT system. Next, we formally define the reference score of a system event, which is a crucial factor in the rareness score.

A. Data Collection

To build the reference model of system events, we collect and compute the statistical data for event occurrences on 54 Linux and 96 Windows machines used daily for product development, research and administration in an enterprise IT system. Particularly, we make special efforts to ensure the representativeness, generality and robustness of the reference model.

1) **Discovery of Homogeneous Hosts***:* The basic idea of reference model is to identify common behaviors across a group of homogeneous hosts. Therefore, to enable this technique, the homogeneity of hosting environment is required; otherwise, the generated model cannot be representative.

In general, enterprise IT systems could satisfy such a requirement due to the overall consistency of daily tasks. However, it is still possible that computers from individual departments in the same corporate carry on different types of workloads, and therefore their system behaviors may vary. To be able to discover the homogeneous groups, we performed a community detection within an enterprise. Particularly, we utilized the Mixed Membership Community and Role model (MMCR) proposed in a prior study [31] and eventually discovered 3 communities within 150 machines. In fact, these 3 communities can be roughly mapped to three different departments in this company. Hence, we collect system events

```
⟨abstract-event⟩ ::= ⟨process-event⟩
                   | ⟨file-event⟩
                   | ⟨network-event⟩
⟨process-event⟩  ::= ⟨process⟩ ⟨process-op⟩ ⟨process⟩
⟨file-event⟩     ::= ⟨process⟩ ⟨file-op⟩ ⟨file⟩
⟨network-event⟩  ::= ⟨process⟩ ⟨network-op⟩ ⟨socket⟩
⟨process⟩        ::= ⟨executable-path⟩
⟨file⟩           ::= ⟨path-name⟩
⟨socket⟩         ::= ⟨remote-address⟩ ':' ⟨remote-port⟩
⟨process-op⟩     ::= 'create'
                   | 'destroy'
⟨file-op⟩        ::= 'read'
                   | 'write'
                   | 'execute'
⟨network-op⟩     ::= 'create'
                   | 'destroy'
                   | 'read'
                   | 'write'
```

Fig. 3: An Abbreviated Syntax of Event Abstraction

from 3 communities separately and build a reference model for each of the detected communities. In this way, the generated models can be adapted for individual environments.

2) **Abstraction of Events***:* To quantify the rareness of system events, our reference model builder expects to count the occurrences of same events. Nonetheless, OS events are highly diverse over time or across hosts, even if they bear the same semantics. For instance, the same program can bear several process IDs when it has been executed for multiple times; two identical system files are assigned with different inode numbers on two Linux hosts.

To capture high-level common behaviors, while tolerating low-level system diversity, we summarize events using their invariant properties. To this end, we first extract semantic-level information from system objects. Particularly, a process is modeled using its executable path, a file is represented by its path name, and a socket is denoted with remote IP address plus remote port number. Then, on top of these representations, we construct the abstraction of events, which follows a grammar illustrated in Figure 3 using Backus-Naur (BNF) form. As a result, events sharing the same abstraction are considered to be same ones.

Note that, due to customization, the path name of same system files may still be different on individual hosts. For example, the user account name can be part of the path name, which in turn becomes unique for each user. To allow such differences, normalization of path name is needed. We address this problem by retrieving a mapping between user account name and the corresponding home directory name from both local machines and global directory services (e.g., active directory, NIS), and replacing the home directory name in the path with the same wildcard.

3) **Time Window***:* The naïve way to count the occurrence of an event is simply increasing the counter, whenever a same one is observed. Nevertheless, this may be subject to poisoning attacks. An adversary can intentionally create repeated malicious activities, and such a burst of vicious events

Week	h1	h2	h3	h4	h5	Total Count
1		x	x			2
2		x		x		4
3	x			x	x	7
4	x	x		x		10
Bit-vector for current week	1	1	0	1	0	

Fig. 4: Computation of Reference Score

may trick the naïve model to believe that these are common behaviors due to their high counts.

To address this problem, we introduce a time window when increasing counters. Within a single time window, repeated occurrence of an event on the same host will only be considered once. As a result, a sudden spike of recurring events only cause limited impacts. We configure the time window to be one week. This is because enterprises are generally operated on weekly basis. Besides, host behaviors within and without work hours, or system activities on weekdays and weekends can be fairly different by nature. Thus, a time window greater than a week can avoid such a vibration of event occurrence while preserving high-level consistency of corporate workloads. Note that the time window is configurable and can be adjusted to different enterprise systems.

B. *Reference Score*

With the aforementioned factors being considered, we formally define the reference score of a system event.

Definition 3. The *Reference Score* ref of an OS-level event e is its accumulative occurrence on all homogeneous hosts for all weeks.

$$ref(e) = \sum_{h \in hosts} \sum_{w \in weeks} count(e, w, h) \tag{3}$$

, where $hosts$ is the set of homogeneous machines, $weeks$ represents the set of weeks when data is collected, and

$$count(e, w, h) = \begin{cases} 1, & \text{if } e \text{ occurred in week } w \text{ on host } h \\ 0, & \text{otherwise} \end{cases}$$

1) **Implementation***:* When computing the score, we in fact update it incrementally using an online algorithm. As depicted in Figure 4, we maintain a total count and a bit-vector of current week for each abstracted event. The bit-vector indicates the occurrence of event on all hosts in the current week, where each bit represents a host. The present data can only affect the existence of event in the current week, and thus will be checked against the bit-vector. By the end of each week, the total count is updated using the bit-vector and the vector will be cleared. In this way, we only store the minimum necessary data so as to ensure efficient storage and query.

V. EVALUATION

In this section, we conduct experiments to evaluate the correctness, effectiveness and efficiency of PRIOTRACKER. First, we present the details of experiment setup. Next, we introduce the metrics and the attack cases used for the evaluations. Then, we provide some insights into the common system operations recorded in the reference model.

A. *Experiment Setup*

We perform all experiments on 54 Linux and 96 Windows machines used daily by researchers, developers and administrators in an enterprise IT environment, with an audit log system capturing OS-level events on host machines. We evaluate PRIOTRACKER on a dataset including 1TB of 2.5 billion events collected from 150 hosts in one week. Our dataset is orders of magnitude larger than the ones used in previous work [9], [20]. ProTracer [9], which is an instrumentation-based tracker, was tested on only 2GB event data; Sleuth [20], a real-time heuristics-based attack graph builder, used merely 20GB data from 6 isolated hosts and did not support cross-host tracking.

To evaluate the correctness and time effectiveness of our approach, we test PRIOTRACKER in eight representative attacks as described in Table I, including data theft, the infamous Shellshock attack, email phishing, Backdoor, as well as attacks and test cases proposed in prior work [9], [25]. The difference is that prior work simply crafted "clean" attack traces isolated from normal system operations. However, in practice, *noise* (i.e., complex and normal system operations) is always interleaved with attack traces due to program logic, shared files and long-running processes. From the daily logs of the enterprise, we observe several typical normal events which connect the malicious activities to benign ones and thus introduce a tremendous amount of noise to the causality graph, and list some examples in Table II. To incorporate the impacts of such noise, we reproduced eight representative attacks in a "noisy" environment where numerous normal activities are considered.

To further verify the time efficiency in real-world causality tracking, we randomly select 75 points of interest (POI), which take excessive time to analyze.

To perform comparative experiments, we also implement a baseline forward-tracker following the prior approach [4], [5], which does not consider priority at all. Instead, this system enables breadth-first search when processing incoming dependencies. We do not use depth-first search because it does not have the capability to escape from a deep branch and thus is usually less effective than breadth-first search.

We run causality trackers on a server equipped with Intel(R) Xeon(R) CPU E5-1650 CPU (12M Cache, 3.20GHz) and 64GB of physical memory. The operating system is Ubuntu 12.04.5 LTS (64bit). Without loss of generality, we only perform forward tracking instead of bi-directional analyses for all the attacks and selected POIs in the experiments. Our reference model is constructed and stored on the same server. Only local I/O overhead is caused when causality tracker queries reference database.

B. *Accuracy*

We use our attack cases to evaluate the accuracy of PRIOTRACKER. To this end, we forward-track all the attacks

TABLE I: Overview of Attack Cases.

Attack Case	Description of Scenario	References
Data Theft	An insider stole sensitive intellectual property, secure-copied the data to a low-profile machine and then leaked it via Internet.	Motivating Example
Phishing Email	A malicious Trojan was downloaded as an Outlook attachment and the enclosed macro was triggered by Excel to create a fake "java.exe", and the malicious "java.exe" further SQL exploited a vulnerable server to start cmd.exe in order to create an info-stealer.	CVE-2008-0081
Shellshock	An attacker utilized an Apache server to trigger the Shellshock vulnerability in Bash multiple times. In each round, she started several Linux commands and cleared Bash history in the end.	CVE-2014-6271
Netcat Backdoor	An attack downloaded the netcat utility and used it to open a Backdoor, from which a port scanner was then downloaded and executed.	Persistent Netcat Backdoor [32]
Cheating Student	A student downloaded midterm scores from Apache, and uploaded a modified version.	ProTracer [9]
Illegal Storage	A server administrator created a directory under another user's home directory, and downloaded the illegal files to the directory.	ProTracer [9]
wget-gcc	Malicious source files were downloaded and then compiled.	Xu et al. [25]
passwd-gzip-scp	An attack stole user account information from passwd file, compressed it using gzip and transferred the data to a remote machine.	Xu et al. [25]

TABLE II: Examples of Normal Activities That Connect the Malicious Activities to Benign Ones and Cause Graph Explosion.

Noise Source	Activity	Reason to Interleave	Description of Scenario
sshd-run-parts	Cascade Forking	Program Behavior	Upon receiving file transferring request, SSH daemon starts a large number of routine processes to update messages (e.g., motd) used for user interaction.
sshd-ypserv	Cascade Forking	Conditional Program Behavior	When a global account requests a SSH connection, SSHD checks user credential from directory service (i.e., NIS) via ypserv, which forks tons of child processes.
.bash_history	Multiple Reads	Shared Log File	Once an attacker has cleared the .bash_history to erase her attack footprints, the same history file will further get read by all future benign Bash terminals.
Explorer	Multiple Writes	Shared GUI Program	Once an attacker has dropped some malware to a local directory, the file Properties are viewed in Explorer by a normal user, who later uncompresses a ZIP file and therefore creates many files using 7ZIP from the same Explorer.

TABLE III: Forward Tracking Results.

Attack Case	Critical Events	Rare CE	Baseline			Prio		
			CE	All	FNR	CE	All	FNR
Data Theft	13	12	13	297	0%	13	297	0%
Phishing Email	148	148	148	3282	0%	148	3282	0%
Shellshock	25	23	25	11252	0%	25	11262	0%
Netcat Backdoor	14	14	14	1355	0%	14	1361	0%
Cheating Student	37	33	14	7526	62%	37	7201	0%
Illegal Storage	12	10	12	8048	0%	12	8201	0%
wget-gcc	25	23	25	6415	0%	25	6742	0%
passwd-gzip-scp	15	11	9	2718	40%	15	2364	0%

via PRIOTRACKER. As a comparison, we also run baseline forward-tracker on the same cases. We run both trackers for one day [1] on each case.

We summarize the results in Table III. Based upon our full knowledge of the attack workflow, we manually collect the ground truth of these attack. Particularly, *critical events* ("CE") indicate the number of essential causal dependencies in an attack. Transitional events such as $bash \rightsquigarrow bash$, which are caused by the underlying system rather than the attack, are not interesting and thus excluded. Among these CEs, we count the ones that occur infrequently ("Rare CE"), i.e., the rareness score is greater than our configured threshold (i.e. $\theta = 0.1$). For the baseline forward-tracker and PRIOTRACKER, We count the CEs and all dependencies discovered within the time limit ("All") and compute the false negative rate ("FNR"), which is the proportion of missed CEs, respectively.

From Table III, we can see that only two incidents ("Data Theft" and "Phishing Email") have the same total number of tracked dependencies for baseline and PRIOTRACKER, indicating that the other incidents cannot complete tracking all causalities within one day, despite the moderate size of attack traces. Nevertheless, we find that PRIOTRACKER captures all the crucial attack causalities within the time limit, whereas the baseline tracker misses 62% and 40% critical events in "Cheating Student" and "passwd-gzip-scp" cases, respectively, due to its intensive computation of noisy dependencies. We also observe that most of CEs are rare, which justifies our choice to prioritize rare events in the causality tracking.

Since all attack cases are conducted in normal noisy enterprise environment, the resulting graphs are up to 600 times larger than essential attack traces. We exemplify the noisy activities that link malicious traces to normal ones in Table II. These noises are interleaved with attack activities fundamentally due to inherent program logic, globally shared logs and central user interfaces that access both malware and benign files.

[1] Note that the time limit for tracking is configurable and can be extended to capture stealthier attackers. We empirically configure their values at this point, and we leave the systematic investigation and optimization of these parameters to future work.

TABLE IV: Elapsed Analysis Time When 100%, 90% and 50% Attack Traces Have Been Revealed, and Average Elapsed Time.

Attack Case	Runtime (100%CEs)		Runtime (90%CEs)		Runtime (50%CEs)		Avg. Runtime for All CEs	
	Baseline	Prio	Baseline	Prio	Baseline	Prio	Baseline	Prio
Data Theft	16.76s	2.62s	14.01s	2.48s	4.03s	2.03s	6.29s	1.55s
Phishing Email	1m2s	1m4s	1m1s	28.48s	1m	28.47s	45.90s	27.51s
Shellshock	2m3s	12.08s	2m2s	11.83s	15.16s	6.48s	37.45s	9.13s
Netcat Backdoor	8.83s	1.28s	8.81s	1.23s	7.01s	0.90s	6.85s	0.88s
Cheating Student	> 1d	40m21s	> 1d	40m19s	> 1d	35m14s	NA	24m47s
Illegal Storage	27m51s	14m10s	26m27s	14m10s	14m3s	13m51s	15m31s	12m39s
wget-gcc	42m9s	6m23s	29m45s	5m54s	12m5s	5m48s	18m31s	5m37s
passwd-gzip-scp	> 1d	1m24s	> 1d	1m23s	3m32s	42s	NA	57s

TABLE V: Ordinal of the Events, Upon Analyzing Which, 100%, 90% and 50% of Attack Traces Have Been Revealed, and Average Ordinal of All Critical Events.

Attack Case	Ordinal (100%CEs)		Ordinal (90%CEs)		Ordinal (50%CEs)		Avg. Ordinal of All CEs	
	Baseline	Prio	Baseline	Prio	Baseline	Prio	Baseline	Prio
Data Theft	174	31	120	27	48	24	60.54	16.38
Phishing Email	3281	3281	1694	244	202	105	1000.83	581.83
Shellshock	11255	35	117	33	65	19	1241.26	23.21
Netcat Backdoor	1347	35	1342	34	529	16	740.69	15.21
Cheating Student	>7526	5481	>7526	5478	>7526	4424	NA	3248.11
Illegal Storage	3971	70	3322	69	1487	58	1742.08	51.42
wget-gcc	4820	610	2473	509	1181	493	1434.64	481.04
passwd-gzip-scp	>2718	210	>2718	208	278	58	NA	98.27

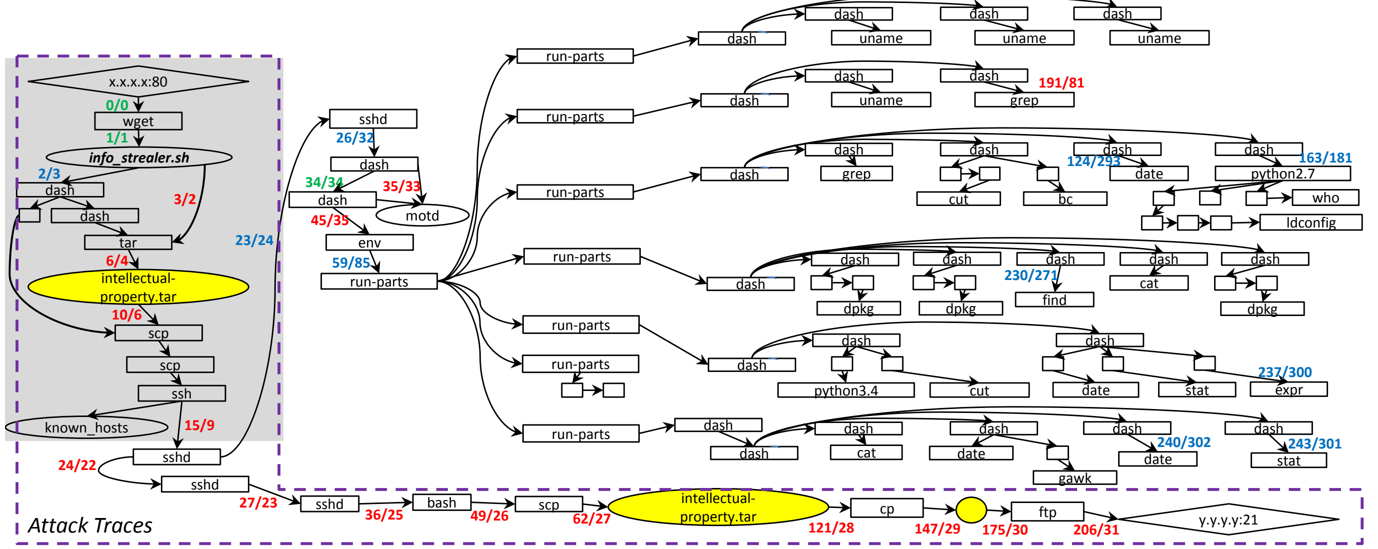

Fig. 5: Reduced Version of Forward Tracking Graph for Motivating Example.

C. Time Effectiveness

We evaluate the time effectiveness of our proposed method. To this end, we examine both baseline tracker and PRIOTRACKER on 1) attack scenarios and 2) randomly selected POIs.

a) Attack Cases: First, we would like to see if PRIOTRACKER can reach attack-related events in a faster manner. Also, we expect to understand in which order these two trackers explore the causality space and whether anomalies can be prioritized by PRIOTRACKER. To this end, we first acquire the total elapsed time to reveal 100%, 90% and 50% critical events for both trackers, and calculate the average elapsed time for reaching every attack-related event. Then, we retrieve the ordinals of all discovered events, compute the average ordinal of all critical events and obtain the ordinal of events, and analyze which 50%, 90% and 100% of attack traces have been uncovered.

As illustrated in Table IV and Table V, our results show that PRIOTRACKER can almost always find all critical events in shorter time within fewer edges. The lower average runtime and ordinals indicate that our tracker can prioritize and stay focused on the exploration and analysis of attack traces. In contrast, baseline tracker may concentrate on attack investigation in the beginning but then quickly lose focus and easily get distracted by environmental noise (i.e., normal system operations). For instance, in the "passwd-gzip-scp" case, the baseline system manages to find 50% of critical events within 3.5 minutes and 278 edges, which are still

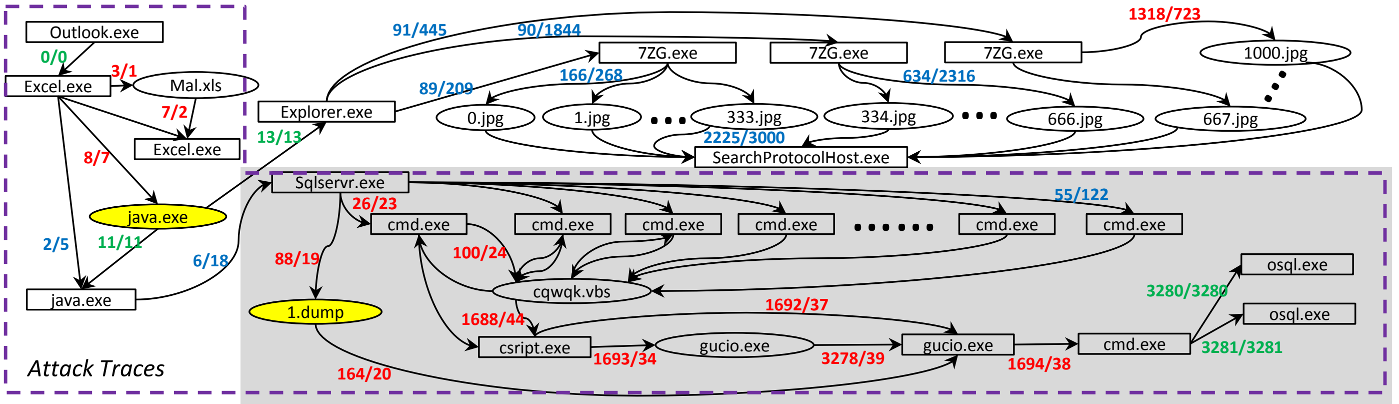

Fig. 6: Reduced Version of Forward Tracking Graph for Email Phishing Attack.

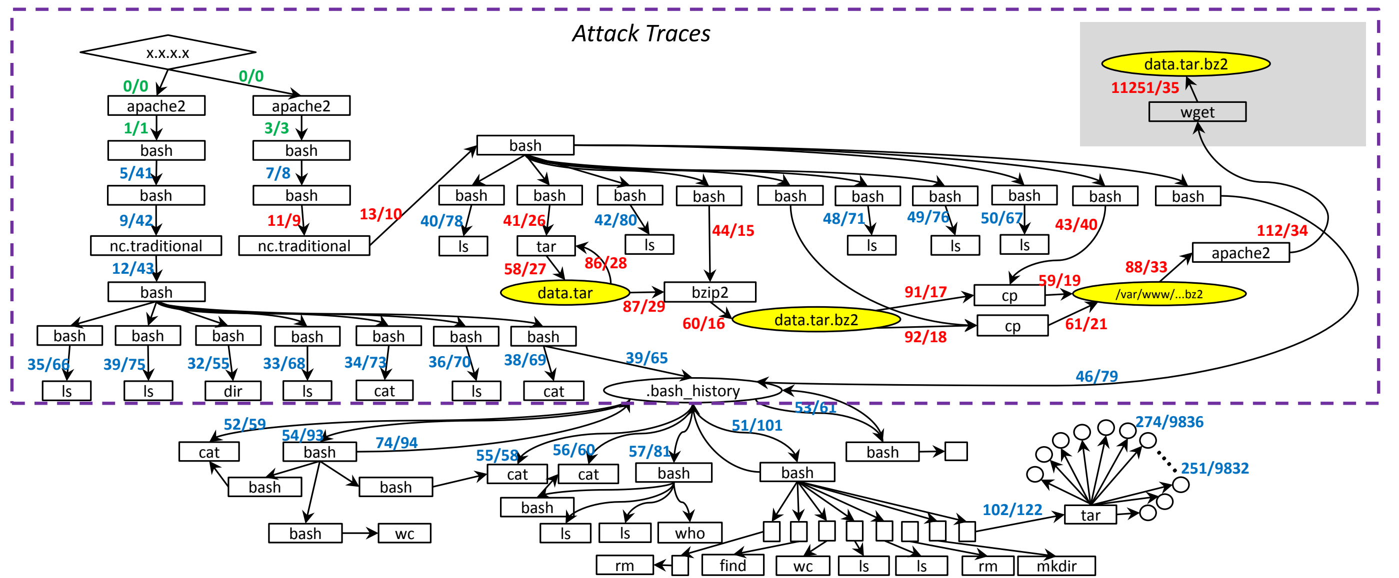

Fig. 7: Reduced Version of Forward Tracking Graph for Shellshock Attack.

comparable to 42s and 58 edges for PRIOTRACKER. However, while PRIOTRACKER eventually finds all attack traces at the 210th edge in 1 minute 24 seconds, baseline tracker cannot reach even 90% of traces within one day or 2718 edges. In the "Phishing Email" attack, however, it takes both trackers the same number of edges to reach full coverage of attack traces, and PRIOTRACKER even spends two more seconds to complete the analysis. This is because this attack ends with a few seemingly common activities, which are not prioritized by our tracker. Nevertheless, PRIOTRACKER can discover the major portion (90%) of this phishing attack twice as fast as the baseline system. Besides, it merely needs 122 edges to cover attack traces in the causality graph, whereas the baseline tracker takes 1400 edges to achieve the same goal.

To gain further insights, we study the visiting order of causalities in PRIOTRACKER and the baseline tracker in three cases, i.e., "Data Theft", "Phishing Email" and "Shellshock". Their attack graphs are presented in Figure 5, Figure 6 and Figure 7, respectively. In these graphs, attack traces are enclosed by dotted lines. Areas with different backgrounds (e.g., grey or white) correspond to different hosts. Interesting edges are labeled with two numbers indicating how many edges are needed to visit them in the baseline tracker and PRIOTRACKER, respectively. We color-code these labels in the presentation: red if the corresponding edge is visited earlier in PRIOTRACKER; blue if it is visited earlier in baseline system; green if it bears the same ordinal using both searching methods.

Data Theft. In the case of motivating example (Figure 5), PRIOTRACKER immediately pursues the direction that leads to the sink of data leakage and can cover all the attack causalities in 31 edges. Our system can focus on attack traces because it prioritizes the transfer of a single new file over the spawn of Linux utilities that are commonly observed. On the contrary, baseline tracker spends analysis time equally on all causalities and therefore jumps back and forth between the attack branch

and normal activities. As a result, for all the attack-related events, PRIOTRACKER outperforms baseline tracker.

Phishing Email. This attack also involved two hosts. As illustrated in Figure 6, once the victim Windows host was compromised by an email attachment containing malicious Macro, a malware program (fake *java.exe*) started and connected to a SQL Server. It then used the SQL Server to create 71 *cmd.exe* processes in order to incrementally craft the malicious script *cqwqk.vbs*. Then, this script file was launched by *cscript.exe*, which thus created another malware instance *gucio.exe*. This malware process issued two SQL commands on SQL Server via *osql.exe* and then received the database dump generated by these commands.

The noise was introduced by accident. Once the fake *java.exe* was written into a directory, a legitimate user opened the same directory using Explorer and unzipped about 1000 files into this folder. The shared GUI programs thus became a bridge for connecting the attack activities to benign events, which hence became *relevant*. Unfortunately, again, baseline tracker lacks the capability of distinguishing anomalies from common behaviors, and thus cannot purely target attack causalities. Due to the frequent switches between attack and noise, the baseline tracker cannot reach specific critical dependencies until 1600 edges later (e.g., $cqwqk.vbs \rightsquigarrow cscript.exe$ or $cscript.exe \rightsquigarrow gucio.exe$).

PRIOTRACKER prefers the dependency $java.exe \rightsquigarrow SqlServr.exe$ to $Explorer.exe \rightsquigarrow 7ZG.exe$ in terms of rareness. Fake *java.exe* is considered special and to be different from authentic Java program because of its unique path. Similarly, our system can quickly scan the majority of attack traces because they are relatively abnormal compared to the 7ZIP program. However, $cmd.exe \rightsquigarrow osql.exe$ is a seemingly common causality, though it is started by the attacker. As a result, PRIOTRACKER decides to traverse the benign file decompression first before it eventually comes back to assess these two events as the 3280th and 3281th edges.

It is worth noting that although the exact events that issue SQL query are not discovered at an early stage, their impacts of dumping and transferring database records have already been captured as the 19th and 20th edges. This indicates that attack footprints are pervasively connected, and partial analysis results are still useful in that they are context clues to infer the existence of other hidden malice, as long as the majority of attack traces are covered. This well motivates PRIOTRACKER which seeks the maximum amount of abnormal activities before analysis deadline.

Shellshock. In this incident, the attacker launched the Bash exploits twice. In the first round, she simply checked environment using Linux utilities without doing serious damage but at the end of this round, she still erased her footprints by clearing Bash history. In the second round, she stole sensitive data. To do this, she archived (*tar*) and compressed (*bzip2*) the files, transferred (*cp*) it to Apache directory so that she can download (*wget*) it from another host. Again, she cleared the history of commands. Later, noises were introduced when a normal user, whose home directory is also /var/www/, opened new Bash terminals. These terminals would read the modified *.bash_history* and forwardly propagated such a causality.

Prior work [4] prunes off all the *.bash_history*s based upon

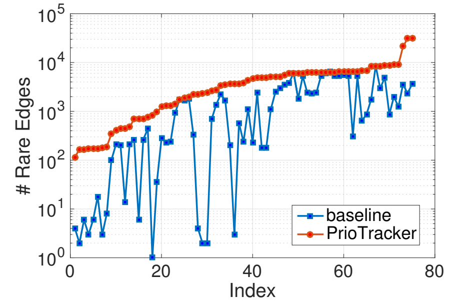

Fig. 8: Number of Rare Edges Discovered Within 1 Hour for 75 Large Graph Testing Samples.

heuristics and does not track any dependencies beyond this shared log file. As a comparison, we address this problem by quantifying the priority of causalities to be analyzed, and therefore can automatically deprioritize the normal Bash activities due to the high fanout of *.bash_history*. In addition, since *.bash_history* is preserved, further dependencies can still be examined at a later stage. In this way, we strike a balance between security and efficiency.

As depicted in Figure 7, PRIOTRACKER prioritizes the investigation of data exfiltration that happened in one of attacker-controlled terminals. It prefers the data leakage path to other launched commands due to two reasons: 1) creation and transfer of new files are more unusual than running Linux utilities; 2) the data exfiltration path can bring significant further consequences while *ls* or *dir* has little future impact. As a result, PRIOTRACKER can find the last hop of attack (i.e., *wget*) after 34 edges. Also, when time permits, PRIOTRACKER eventually studies all the normal causalities, such as data decompression (*tar*) after around 9800 edges.

In general, since the baseline tracker spends time evenly on all branches, as the tracking progresses and an exponentially growing number of dependencies get involved, the baseline tracker cannot reach the deepest stage of attack before going through all normal activities. PRIOTRACKER prioritizes the investigation of anomalies and automatically find the major attack traces in a much more effective and timely manner.

1) Random POI: Next, we hope to understand if PRIOTRACKER can identify the maximum amount of uncommon events within a certain time limit (configured to be one hour in our training phase). To this end, we run PRIOTRACKER and baseline tracker on 75 randomly selected POI which on average can generate a forward tracking graph of 5,732 edges.

Figure 8 illustrates the number of rare edges for 75 cases, captured by both trackers within one hour. Here, a"rare" edge bears a rareness score that is greater than the threshold θ we set. As depicted in the chart, the red curve representing our result is always above the blue one for baseline tracker, while in certain cases, PRIOTRACKER can show up to three orders of magnitude improvement. This also justifies the effectiveness of our priority computation and weight selection.

D. Insight into Reference Model

1) Runtime Overhead: PRIOTRACKER performs two additional queries to compute the priority scores: 1) look-ahead

query and 2) reference model query. Look-ahead query does not introduce additional runtime overhead in the long run because query results are cached for future dependency construction. Reference database query, in contrast, may cause additional I/O overhead, and therefore needs evaluation. To this end, we record the elapsed time of each query to reference database. Our result shows that every access merely causes negligible slowdown, on average 0.95 microseconds.

2) Case Study: We then look into the content of our reference model and attempt to understand its validity based upon case studies.

Process Creation. Many models are generated because of common "forking" relation between parent and child processes. Particularly, the top ones are caused by mutual invocation of system processes. For instance, in Windows, *services.exe* is the parent of multiple service programs, such as *taskhost.exe* and *conhost.exe*; in Linux, it is common to see *bash* forking other terminals, *bash* and *dash*, or utilities such as *cat*, *grep*, *date*, etc. Common user programs, such as *python2.7*, *acrobat.exe*, *iexplorer.exe*, *outlook.exe* and *chrome.exe*, may also bear regular invocation behaviors. Program updaters, including *dpkg*, *apt-get*, *googleupdate.exe* and especially antivirus updater, *sesclu.exe* (Symantec Endpoint Security Client LiveUpdate), frequently start subroutines to acquire new packages manually or automatically.

File Access. A even greater amount of models have been generated from file accesses. However, they become part of our model not because the files are popular ones that commonly read/written by different processes. On the contrary, they become prevalent mainly due to the popularity of the processes. For example, on Windows, system processes, such as *svchost.exe*, *services.exe*, *taskhost.exe*, *system.exe* frequently access their libraries, logs, metadata, configurations, font etc; antivirus scanners *coh64.exe* also accesses a list of specific files for detection and management purposes. On the Linux side, *man-db* utility may repeatedly access its own database, indexes and caches, while *updatedb.mlocate* keeps updating these cache files; *apt-get* reads its configuration very often; *find* frequently accesses its cached records; *python2.7* loads functions from a list its internal libraries.

File Execution. File execution is a special class of read operation and bears completely different semantics. Especially, due to the fundamental difference in design and implementation, file execution on Windows is very different from that on Linux: most of the executed Windows files are .DLL files, which are dynamically loaded by various of system processes, while Linux usually directly forks a process from an individual executable file.

VI. Discussion

1) Context Sensitivity of Reference Model: In this paper, we only consider the rareness of one event edge. However, given an event edge, the nature of tracking provides the knowledge of subsequent events in backtracking and preceding events in forwardtracking, which can form the context of this event and be leveraged to differentiate attack-related activities from normal ones more accurately. Given an event, priority scores can take its context, i.e., rareness of event edges within its k-hop neighborhood, into consideration. We will leave it to future work.

2) Adversarial Setting: Reference scores are naturally resilient to poisoning attacks, since repeated occurrences of an event on the same host in a week are only considered once. APT adversaries has limited power in compromising a sufficient number of hosts and performing the same malicious behaviors across weeks, which can lead to higher risk of being detected.

2a) Evasion Using High-fanout Events: We admit that, as a potential evasion technique, an attacker may attempt to leverage system causality with high fanout to hide their attack footprints. However, an attack cannot be launched solely using dependencies with big fanout. For instance, $apache \rightsquigarrow bash$ is an essential low-fanout step when a web server gets compromised. Other attack-related edges can be discovered by our approach from thousands of benign edges in a faster fashion. Since the entire attack footprints are logically connected, any uncovered part can help analysts to identify the other parts. Even if, in the worst case scenario, fast-tracking an event with low fanout does not expose any attack traces, only a small delay will be incurred to the investigation of other complex causalities. On the contrary, processing benign dependencies with huge fanout (up to tens of thousands) can be time consuming such that none of attack traces can be reached before analysis deadline.

2b) Evasion Using Low-fanout Events: An attacker may intentionally fill the priority queue with numerous benign low-fanout events to conceal the later steps in an attack. However, due to the low-fanout constraint, she has to craft a sufficiently long chain of low-fanout events, which by itself is a extremely suspicious topological pattern. As a result, our technique significantly raises the bar for potential attacks.

2c) Slow Attack: A slow attacker is a general challenge for all causality tracking systems but may also risk being detected even before she can cause serious damage, as defenders could also take advantage of the longer time window.

Note that our work is a general framework that prioritizes abnormal activities for timely security causality analysis, which is able to further incorporate multi-hop dependency based attack patterns or user defined priority scores customized for specific environments and security requirements.

3) Distributed Causality Tracker: The construction of causality graphs can be potentially parallelized with distributed computing. Any individual branch to be explored can be processed separately; branches may bear different priorities and therefore are assigned with corresponding computing resources; dependencies on each host can also be pre-computed in parallel and cross-host tracking thus becomes the concatenation of multiple generated graphs. Nonetheless, the massive and pervasive dependencies among system events bring significant challenges to parallel processing, and therefore distributed causality tracking by itself is an interesting research direction that requires non-trivial efforts.

In this work, however, we do not enable distributed computation in our causality tracking. Instead, we retrieve audit logs from multiple hosts, store them in a centralized database, and then perform causality analysis in a centralized manner.

Hence, our analysis only generates one single holistic graph to demonstrate an attack sequence even if the attack is across multiple hosts. Cross-host tracking is conducted in an on-demand manner: only if the causality tracker discovers a communication channel from a sender machine to a receiver, it will start to build dependencies on the latter one.

Again, we would like to point out that the major focus of our work is how to enable priority-based search in causality tracking, which is orthogonal to the computing paradigms of underlying tracking systems.

VII. Related Work

1) Causality Analysis: Plenty of research efforts have been made to reconstruct OS-level system dependencies for security purposes. King and Chen [4] first proposed to build dependency graph based on OS-level system events in order to capture the attack sequences and provenances. King et al. [5] further improved the dependency analysis by enabling cross-host tracking as well as forwardly tracking attack consequences. Chow et al. [33] leveraged taint analysis to understand the lifetime of sensitive data. Taser [10] also performed taint tracking to find files affected by a past attack. Retro [11] recorded an action history graph, which describes system's execution, in order to repair a desktop or server after an adversary compromises it. Jiang et al. [34] enabled a provenance-aware tracing of worm break-in and contamination. Muniswamy-Reddy et al. [35] designed a provenance collection structure facilitating the integration of provenance across multiple levels of abstraction. Krishnan et al. [36] provided a forensic platform that transparently monitors and records data access events using only the abstractions exposed by the hypervisor. Hi-Fi [37] presented a kernel-level provenance system that collects high-fidelity whole-system provenance. Ma et al. [8] proposed a Windows based audit logging technique that features accuracy and low cost.

Further studies have attempted to mitigate the dependency explosion problem by reducing data volume and performing fine-grained causality tracking. BEEP [6] identified the event handling loops in long running programs so as to enable selective logging for unit boundaries and unit dependencies. LogGC [7] proposed an audit logging system with garbage collection capability. ProTracer [9] presented a lightweight provenance tracing system that alternates between logging and taint tracking. Xu et al. [38] attempted to reduce the number of log entries while still preserving high-fidelity causal dependencies.

A recent line of research [28], [39] has enabled enterprise-level causality analysis, such as data loss prevention, via modifying underlying operating system. To this end, it introduces Linux Provenance Modules, which produce fine-grained provenance information. It further mitigates the dependency explosion using SELinux information flow analysis which removes unrelated provenances.

Compared to the prior work, PRIOTRACKER takes the first step to prioritizing the investigation of abnormal dependencies during the construction of causality graph. As a result, the subsequent causality tracking can reveal more unusual activities before a critical security analysis deadline. In contrast, the previous work did not innovate new algorithms for attack graph construction but rather followed original work [4], [5] to generate causality graphs via simply traversing all nodes (both normal and abnormal). PRIOTRACKER focuses on improving the underlying graph generation algorithm, and therefore is orthogonal to the prior research.

2) Priority-Based Security Analysis: Priority-based methods have been widely used in security analyses. Previous efforts have been made to expedite static data-flow analysis [13], symbolic execution [14]–[16], fuzzing [17] and digital forensics [18], [19]. To be able to prioritize certain tasks, these studies have attempted to measure the priority of low-level constructs, including functions, code paths, program inputs or user-level entities, such as textual documents and physical devices.

In contrast to the prior work, we enable a priority-based analysis in the specific domain of causality tracking. As a result, we have to invent a unique technique to quantify the priority in OS-level dependency tracking.

VIII. Conclusion

In this paper, we propose PRIOTRACKER, a backward and forward causality tracker that automatically prioritizes the investigation of abnormal causal dependencies for enterprise security. Specifically, to assess the priority of a system event, we consider its rareness and topological features in the causality graph. To distinguish unusual operations from normal system events, we quantify the rareness of each event by building a reference model which records common routine activities in corporate computer systems. We implement PRIOTRACKER, in 20K lines of Java code, and a reference model builder in 10K lines of Java code. We evaluate our tool by deploying both systems in a real enterprise IT environment which consists of 150 machines. Experimental results show that PRIOTRACKER can capture attack traces that are missed by existing trackers and reduce the analysis time by up to two orders of magnitude.

Acknowledgment

We would like to thank anonymous reviewers and our shepherd, Prof. Adam Bates, for their feedback in finalizing this paper. Prateek Mittal and Yushan Liu were partially supported by the National Science Foundation Grant CNS-1553437, CIF-1617286, and CNS-1409415, and Yan Huo *94 Graduate Fellowship. Any opinions, findings, and conclusions made in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] "Data breaches," http://www.idtheftcenter.org/Data-Breaches/data-breaches.html, 2016.

[2] C. Staff, "Target: 40 million credit cards compromised," http://money.cnn.com/2013/12/18/news/companies/target-credit-card/, 2013.

[3] R. Sidel, "Home depot's 56 million card breach bigger than target's," http://www.wsj.com/articles/home-depot-breach-bigger-than-targets-1411073571, 2014.

[4] S. T. King and P. M. Chen, "Backtracking Intrusions," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP'03, 2003.

[5] S. King, Z. M. Mao, D. C. Lucchetti, and P. M. Chen, "Enriching Intrusion Alerts Through Multi-Host Causality," in *Proceedings of the 2005 Network and Distributed Systems Security Symposium*, ser. NDSS'05, 2005.

[6] K. H. Lee, X. Zhang, and D. Xu, “High Accuracy Attack Provenance via Binary-based Execution Partition,” in *Proceedings of the 2013 Network and Distributed Systems Security Symposium*, ser. NDSS'13, 2013.

[7] ——, “Loggc: garbage collecting audit log,” in *Proceedings of the 2013 ACM SIGSAC conference on Computer and Communications Security*, ser. CCS'13, 2013.

[8] S. Ma, K. H. Lee, C. H. Kim, J. Rhee, X. Zhang, and D. Xu, “Accurate, Low Cost and Instrumentation-Free Security Audit Logging for Windows,” in *Proceedings of the 31st Annual Computer Security Applications Conference*, ser. ACSAC'15, 2015.

[9] S. Ma, X. Zhang, and D. Xu, “Protracer: Towards practical provenance tracing by alternating between logging and tainting.” in *Proceedings of the 2016 Network and Distributed Systems Security Symposium*, ser. NDSS'16, 2016.

[10] A. Goel, K. Po, K. Farhadi, Z. Li, and E. de Lara, “The Taser Intrusion Recovery System,” in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, ser. SOSP'05, 2005.

[11] T. Kim, X. Wang, N. Zeldovich, and M. F. Kaashoek, “Intrusion recovery using selective re-execution,” in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10, 2010.

[12] “2015 cost of cyber crime study: United states,” http://www.ponemon.org/blog/2015-cost-of-cyber-crime-united-states, 2015.

[13] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, “Taj: Effective taint analysis of web applications,” in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09, 2009.

[14] V. Chipounov, V. Kuznetsov, and G. Candea, “S2e: A platform for in-vivo multi-path analysis of software systems,” in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI, 2011.

[15] C. Cadar, D. Dunbar, and D. Engler, “Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs,” in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08, 2008.

[16] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, “Unleashing mayhem on binary code,” in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, ser. SP '12, 2012.

[17] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, “Driller: Augmenting fuzzing through selective symbolic execution,” in *23nd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, 2016.

[18] R. Bert, F. Marturana, G. Me, and S. Tacconi, “Data mining based crime-dependent triage in digital forensics analysis,” in *Proceedings of 2012 International Conference on Affective Computing and Intelligent Interaction*, 2012.

[19] N. L. Beebe and L. Liu, “Ranking algorithms for digital forensic string search hits,” *Digit. Investig.*, 2014.

[20] M. N. Hossain, S. M. Milajerdi, J. Wang, B. Eshete, R. Gjomemo, R. Sekar, S. Stoller, and V. Venkatakrishnan, “Sleuth: Real-time attack scenario reconstruction from cots audit data,” 2017.

[21] “The seven largest insider-caused data breaches of 2014,” http://www.eweek.com/security/the-seven-largest-insider-caused-data-breaches-of-2014, 2014.

[22] “Indian call centers selling u.k.'s secrets,” http://www.siliconindia.com/shownews/Indian_call_centers_selling_UKs_secrets-nid-28560-cid-2.html, 2005.

[23] “Understanding the insider threat,” https://supportforums.cisco.com/blog/150466/understanding-insider-threat, 2016.

[24] “Incident response - time is of the essence,” https://www.scmagazineuk.com/incident-response--time-is-of-the-essence/article/534765/, 2015.

[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 *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, 2016.

[26] “The linux audit framework,” https://www.suse.com/documentation/sled10/audit_sp1/data/book_sle_audit.html, 2016.

[27] “Etw events in the common language runtime,” https://msdn.microsoft.com/en-us/library/ff357719(v=vs.110).aspx, 2016.

[28] A. M. Bates, D. Tian, K. R. Butler, and T. Moyer, “Trustworthy whole-system provenance for the linux kernel.” in *USENIX Security Symposium*, pp. 319–334.

[29] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2003.

[30] “A persistent key-value store for fast storage environments,” http://rocksdb.org/, 2016.

[31] T. Chen, L.-A. Tang, Y. Sun, Z. Chen, H. Chen, and G. Jiang, “Integrating Community and Role Detection in Information Networks,” in *Proceedings of 2016 SIAM International Conference on Data Mining (SDM'16)*, 2016.

[32] “Persistent netcat backdoor,” https://www.offensive-security.com/metasploit-unleashed/persistent-netcat-backdoor/, 2017.

[33] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum, “Understanding data lifetime via whole system simulation,” in *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, 2004.

[34] X. Jiang, A. Walters, F. Buchholz, D. Xu, Y.-M. Wang, and E. H. Spafford, “Provenance-Aware Tracing of Worm Break-in and Contaminations: A Process Coloring Approach,” in *Proceedings of IEEE ICDCS06*, 2006.

[35] K.-K. Muniswamy-Reddy, U. Braun, D. A. Holland, P. Macko, D. Maclean, D. Margo, M. Seltzer, and R. Smogor, “Layering in provenance systems,” in *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, ser. USENIX'09, 2009.

[36] S. Krishnan, K. Z. Snow, and F. Monrose, “Trail of bytes: Efficient support for forensic analysis,” in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, ser. CCS '10, 2010.

[37] D. J. Pohly, S. McLaughlin, P. McDaniel, and K. Butler, “Hi-fi: Collecting high-fidelity whole-system provenance,” in *Proceedings of the 28th Annual Computer Security Applications Conference*, ser. ACSAC '12, 2012.

[38] 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 *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS'16, 2016.

[39] A. Bates, D. J. Tian, G. Hernandez, T. Moyer, K. R. Butler, and T. Jaeger, “Taming the costs of trustworthy provenance through policy reduction,” *ACM Transactions on Internet Technology (TOIT)*, vol. 17, no. 4, p. 34, 2017.