NODOZE: Combatting Threat Alert Fatigue with Automated Provenance Triage

Wajih Ul Hassan[◇], Shengjian Guo[‡], Ding Li[*], Zhengzhang Chen[*], Kangkook Jee[*], Zhichun Li[*], Adam Bates[◇]

[◇] *University of Illinois at Urbana-Champaign*
{whassan3,batesa}@illinois.edu

[‡] *Virginia Tech*
guosj@vt.edu

[*] *NEC Laboratories America, Inc.*
{dingli,zchen,kjee,zhichun}@nec-labs.com

***Abstract*—Large enterprises are increasingly relying on threat detection softwares (e.g., Intrusion Detection Systems) to allow them to spot suspicious activities. These softwares generate alerts which must be investigated by cyber analysts to figure out if they are true attacks. Unfortunately, in practice, there are more alerts than cyber analysts can properly investigate. This leads to a "threat alert fatigue" or information overload problem where cyber analysts miss true attack alerts in the noise of false alarms.**

In this paper, we present NODOZE to combat this challenge using contextual and historical information of generated threat alert. NODOZE first generates a causal dependency graph of an alert event. Then, it assigns an anomaly score to each edge in the dependency graph based on the frequency with which related events have happened before in the enterprise. NODOZE then propagates those scores along the neighboring edges of the graph using a novel network diffusion algorithm and generates an aggregate anomaly score which is used for triaging. We deployed and evaluated NODOZE at NEC Labs America. Evaluation on our dataset of 364 threat alerts shows that NODOZE consistently ranked the true alerts higher than the false alerts based on aggregate anomaly scores. Further, through the introduction of a cutoff threshold for anomaly scores, we estimate that our system decreases the volume of false alarms by 84%, saving analysts' more than 90 hours of investigation time per week. NODOZE generates alert dependency graphs that are two orders of magnitude smaller than those generated by traditional tools without sacrificing the vital information needed for the investigation. Our system has a low average runtime overhead and can be deployed with any threat detection software.

I. INTRODUCTION

Large enterprises are increasingly being targeted by *Advanced Persistent Threats* (APTs). To combat these threats, enterprises are deploying threat detection softwares (TDS) such as intrusion detection system and security information and event management (SIEM) tools. These softwares constantly monitor the enterprise-wide activities and generate a threat alert if a suspicious activity happens. Cyber analysts then manually sift through these alerts to find a signal that indicates a true attack.

Network and Distributed Systems Security (NDSS) Symposium 2019
24-27 February 2019, San Diego, CA, USA
ISBN 1-891562-55-X
https://dx.doi.org/10.14722/ndss.2019.23349
www.ndss-symposium.org

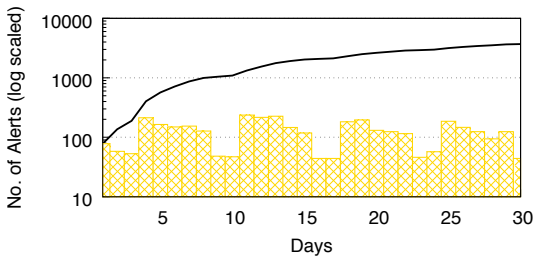


Fig. 1: Growth of alerts in an enterprise during a given month.

Unfortunately, these automated systems are notorious for generating high rates of false alarms [59], [2], [6]. According to a recent study conducted by FireEye, most organizations receive 17,000 alerts per week where more than 51% of the alerts are false positives and only 4% of the alerts get properly investigated [4]. Due to an enormous number of alerts, cyber analyst face "threat alert fatigue"[1] problem and important alerts get lost in the noise of unimportant alerts, allowing attacks to breach the security of the enterprise. One example of this is Target's disastrous 2013 data breach [15], when 40 million card records were stolen. Despite numerous alerts, the staff at Target did not react to this threat in time because similar alerts were commonplace and the security team incorrectly classified them as false positives. In Fig. 1, we demonstrate the growth of alerts generated by a commercial TDS [8] at NEC Labs America comprising 191 hosts.

The threat alert fatigue problem is, at least partially, caused by the fact that existing academic [43], [29] and commercial [3], [5] TDS use heuristics or approaches based on single event matching such as an anomalous process execution event to generate an alert. Unfortunately, in many cases, a false alert may look very similar to true alert if the investigator only checks a single event. For example, since both ransomware and `ZIP` programs read and write many files in a short period of time, a simple ransomware detector that only checks the behavior of a single process can easily classify `ZIP` as ransomware [40]. Even though contextual alerting has proven to be most effective in the alert triage process [27], existing TDS usually do not provide enough contextual information about alerts (*e.g.*, entry point of invasion) which also increases investigators' mean-time-to-know.[2]

[1] A phenomenon when cyber analysts do not respond to threat alerts because they receive so many each day.

[2] Mean-time-to-know measures how fast cyber analysts can sort true threats from noise when they get threat alerts.

Data provenance analysis [41], [26] is one possible remedy for the threat alert fatigue problem. Data provenance can provide the contextual information about the generated alert through reconstructing the chain of events that lead to an alert event (backward tracing) and the ramifications of the alert event (forward tracing). Such knowledge can better separate a benign system event from a malicious event even though they may look very similar when viewed in isolation. For example, by considering the provenance of an alert event, it is possible to distinguish ransomware from ZIP: the entry point of ransomware (*e.g.*, email attachment) is different from the ZIP program.

Although a provenance-based approach sounds promising, leveraging data provenance for triaging alerts suffers from two critical limitations: *1) labor intensive* – using existing techniques still require a cyber analyst to manually evaluate provenance data of each alert in order to eliminate false alarms, and *2) dependency explosion problem* – due to the complexity of modern system, current provenance tracking techniques will include false dependencies because an output event is assumed to be causally dependent on all preceding input events [46]. In our scenario, due to this problem, a dependency graph of a true attack alert will include dependencies with benign events which might not be causally related to the attack. This problem makes the graph very huge (with thousands or even millions of nodes). Such a huge graph is very hard for security experts to understand [36], making the diagnosis of attacks prohibitively difficult.

In this paper, we propose NODOZE, an automatic alert triage and investigation system based on provenance graph analysis. NODOZE leverages the historical context to automatically reduce the false alert rate of existing TDS. NODOZE achieves this by addressing the aforementioned two limitations of existing provenance analysis techniques: it is fully automated and can substantially reduce the size of the dependency graphs while keeping the true attack scenarios. Such concise dependency graphs enable security experts to better understand the attacks, discover vulnerabilities quickly, accelerating incident response.

Our approach is based on the insight that *the suspiciousness of each event in the provenance graph should be adjusted based on the suspiciousness of neighboring events in the graph.* A process created by another suspicious process is more suspicious than a process created by a benign process. To this end, our anomaly score assignment algorithm is an unsupervised algorithm with no training phase. To assign anomaly scores to the events, NODOZE builds an Event Frequency Database which stores the frequencies of all the events that have happened before in the enterprise. After anomaly score assignment, NODOZE uses a novel network diffusion algorithm to efficiently propagate and aggregate the scores along the neighboring edges (events) of the alert dependency graph. Finally, it generates an aggregate anomaly score for the candidate alert which is used for triaging.

To tackle the dependency explosion problem in the alert investigation process, we propose the notion of *behavioural execution partitioning*. The idea is to partition a program execution based on normal and anomalous behaviour and generate most anomalous dependency graph of a true alert. This allows cyber analyst to focus on most anomalous events which are causally related to the true alert which accelerates the alert investigation process.

We implement NODOZE and event frequency database in 9K and 4K lines of Java code respectively. We deployed and evaluated our system at NEC Labs America. For evaluation we used 1 billion system events spanning 5 days which generated 364 alerts using an exemplar TDS [8]. These alerts include 10 APT attack cases and 40 recent malware simulation while all the other alerts are false alarms. Experimental results show that NODOZE improves the accuracy of existing TDS by reducing the false alarms by 84%. Moreover, NODOZE generates dependency graphs for true alerts that are *two orders of magnitude* smaller than those generated by traditional tools.

In summary, this paper makes the following contributions:

- We propose NODOZE, an automated threat alert triage system for enterprise settings.
- We present a novel network diffusion algorithm to propagate anomaly scores in dependency graphs enabling the calculation of aggregate anomaly scores for threat alerts.
- We introduce the notion of *behavioural execution partitioning*, a new technique for combating dependency explosion in provenance graph that is applicable to threat alerts.
- We present a concrete implementation and thorough evaluation of NODOZE. The results show that NODOZE consistently ranked the true alerts higher than false alarms and generates concise dependency graphs for true alerts.

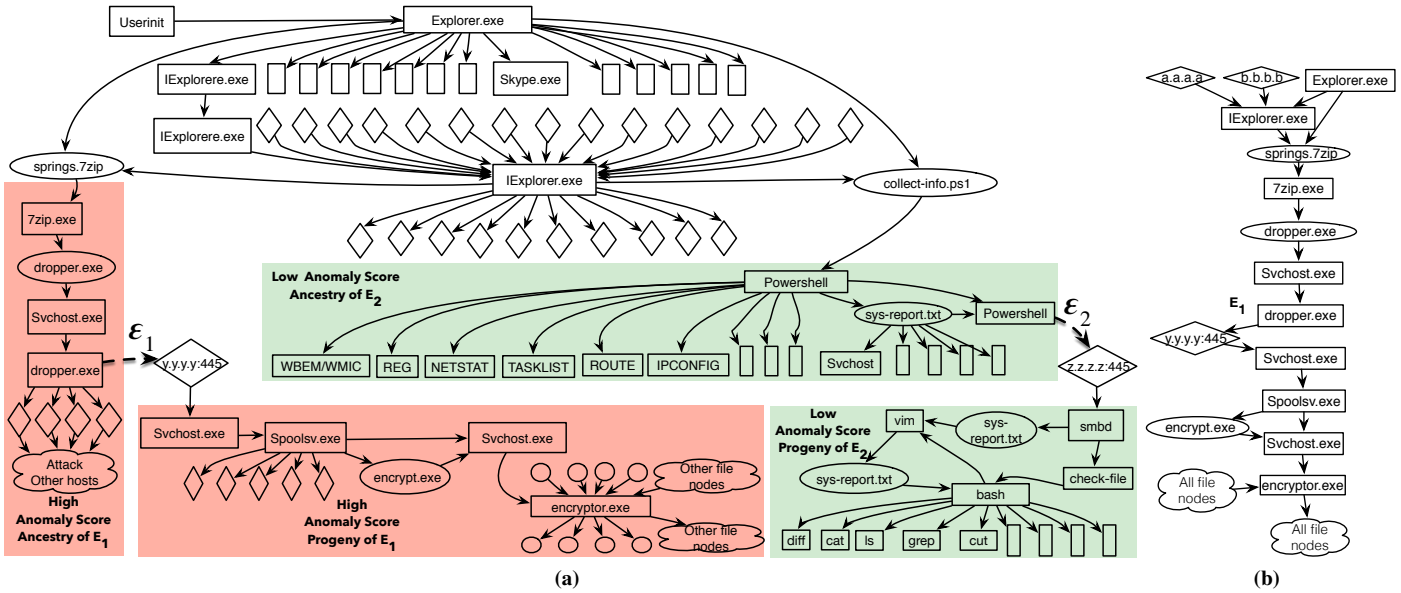
II. BACKGROUND & MOTIVATION

In this section, we use an attack example to illustrate the effectiveness and utility of NODOZE as an alert triage system with two aspects: 1) filtering out false alarms to reduce alert fatigue, and 2) concise explanation of the true alerts using dependency graphs to accelerate alert investigation process. We will use the example of a WannaCry ransomware attack [18] in an enterprise environment. This attack was simulated as a live exercise at NEC Labs America; we describe the experimental setup used for the simulation in §VIII.

A. Motivating Attack Example

WannaCry ransomware is a popular attack that affected around 0.2 million systems across 150 countries in May 2017 [12]. It is essentially a cryptoworm which targets computers running the Microsoft Windows OS with vulnerable EternalBlue [14]. It exploits this vulnerability to gain access to the machines and encrypts data on those machines.

Scenario. Consider a front desk person in an enterprise who one day visits several websites using Internet Explorer to search for pdf reader software. After visiting several links, the front desk person accidentally downloads a malware (`springs.7zip`) from a malicious website and then runs the malware thinking of it as pdf reader software. This malware opens a backdoor to the attacker's server and then searches for EternalBlue vulnerable machines in the front desk's enterprise network. Once vulnerable machines are found the attacker downloads the file encryptor and starts to encrypt files on those vulnerable machines. After some time the front desk person's PC starts to run very slow so front desk person calls technical



(a) (b)

Fig. 2: WannaCry attack scenario described in §II-A. **(a)** Part of the threat alerts' dependency graph generated by prior approaches [26], [41]. Some edges have been omitted for clarity. **(b)** Concise dependency graph generated by NODOZE.

support. The technical support person downloads and executes a diagnostic tool (`collect-info.ps1`) on front desk person's PC from an internal software repository, which runs some diagnostic commands including `Tasklist` and `Ipconfig`. All of the output is copied to a file `sys-report.txt`, which is then transferred to a remote machine for further investigation. On the remote machine, the technical support person runs several `bash` commands to check the file contents and figure out the issue with the front desk person's computer.

Alerts Investigation. During the above attack scenario, two threat alerts were generated by the underlying TDS while over 100 total threat alerts were generated over the course of the day. The first alert event $\mathcal{E}_1$, was generated when malware made several connections to remote machines in the enterprise. The second alert event $\mathcal{E}_2$ was generated when technical support diagnostic tool initiated a remote connection to a secure machine. Note that, at a single event level, both alert events $\mathcal{E}_1$ and $\mathcal{E}_2$ look very similar; both processes making an unusual connection to a remote machine in the network.

To investigate the alerts and prepare a response, the cyber analyst performs a causality analysis. Provenance-based tools [41], [26] process individual events between system objects (*e.g.*,, files and network sockets) and subjects (*e.g.*,, processes) to construct a causal dependency graph. Note that cyber analysts can use these graphs to understand the context of the alert by using a backward tracing query which starts from the given symptom event (alert) and then identifies all the subjects and objects that the symptom directly and indirectly depends on. Using a *forward tracing* query, the analyst can then identifies all the effects induced by the root cause of the alert. Fig. 2a shows the simplified dependency graph generated by existing tools for alert events $\mathcal{E}_1$ and $\mathcal{E}_2$. In this graph and also the rest of the paper, we use *diamonds*, *ovals*, *boxes*, and *dashed arrows* to represent sockets, files, processes, and alert events respectively.

B. Existing Tools Limitations

Existing provenance trackers when combined with TDS for alert triage and investigation process suffer from following limitations:

Alert Explosion & Manual Labor. Even if the TDS identifies an anomalous event related to the attack, cyber analysts are barraged with alerts on a daily basis and face the problem of finding a "needle in a haystack". Existing automated TDS are notorious for generating a high amount of false alarms [59], [2], [6], [34], [21]. Cyber analysts are in short supply, so organizations face a key challenge in managing the enormous volume of alerts they receive using the limited time of analysts [4]. Many heuristic- and rule-based static approaches have been proposed to mitigate this problem [68], [22], [45], [32]. However, there are still too many threat alerts for the analysts to manually investigate in sufficient depth using alerts' dependency graphs which are also usually very complex. During the day of the attack, the TDS generated over 100 threat alerts with an average of 2K vertices in each alert's dependency graph; and only 1 threat alert was related to WannaCry attack while all other were false alarms.

Dependency Explosion. Most existing provenance trackers suffer from the *dependency explosion problem*, generating graphs similar to Fig. 2a. The dependency inaccuracy is mainly caused by long running processes that interact with many subjects/objects during their lifetime. Existing approaches consider the entire process execution as a single node so that all input/output interactions become edges to/from the process node. This results in considerably large and inaccurate graphs. Consider the Internet Explorer `IExplorer.exe` vertex in our example dependency graph which is shown in Fig. 2a. When cyber analysts try to find the ancestry of the downloaded malware file (`springs.7zip`) and diagnostic tool file (`collect-info.ps1`), they will unable to determine which incoming IP/socket connection vertex is related to the malware file and which one belongs to the diagnostic tool file.

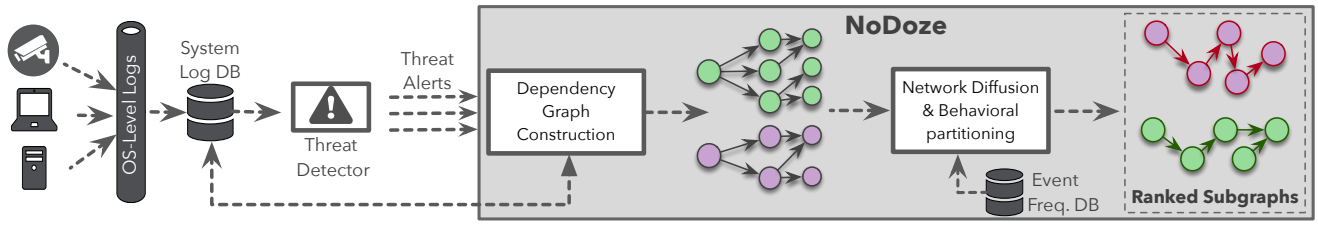


Fig. 3: Overview of NODOZE. Alerts generated by threat detector are provided to NODOZE, which ranks the alerts based on their aggregate anomaly scores and produces concise alert dependency graphs for investigation.

Prior solutions to the dependency explosion problem [46], [51], [50], [44] propose to partition the execution of a long running process into autonomous "units" in order to provide more precise causal dependency between input and output events. However, these systems require end-user involvement and system changes through source code instrumentation, training runs of application with typical workloads, and modifying the kernel. Due to proprietary software and licensing agreements, code instrumentation is not often possible in an enterprise. Furthermore, these systems are only implemented for Linux, and their designs are inapplicable to commodity-off-the-shelf operating systems like Microsoft Windows. Finally, acquiring typical application workloads in a heterogeneous large enterprise is not practically feasible.

C. Goals

The aforementioned limitations motivate the following design goals for the NODOZE system:

- *Alert Reduction.* NODOZE should reduce false positives, false negatives and non-actionable items as compared to existing TDS.
- *Concise Contextual Alerting.* NODOZE-generated dependency graphs of threat alerts should be concise and complete.
- *Generality.* NODOZE design and techniques should be independent of underlying platforms (e.g. OS, VM, etc.), applications, and TDS.
- *Applicability.* NODOZE should not require any end system change and should be deployable on any existing TDS.

III. NODOZE OVERVIEW & APPROACH

The overall workflow of NODOZE system to triage alerts based on anomaly scores is shown in Fig. 3. NODOZE acts as an add-on to an existing TDS in order to reduce false alarms and provide contextual explanations of generated threat alerts. To triage alerts, NODOZE first assigns an anomaly score to each event in the generated alerts provenance graph. Anomaly scores are calculated using frequencies with which related events have happened before in the enterprise. NODOZE then uses a novel network diffusion algorithm to propagate and aggregate anomaly scores along the neighboring events. Finally, it generates an aggregate anomaly score for the generated alert which is used for triaging – escalating the most critical incidents for remediation and response.

As mentioned previously, existing execution partitioning techniques [46], [51], [50], [44] for precise dependencies are not feasible in an enterprise. In the case of true alerts, NODOZE solves this problem by leveraging the observation that the attack's dependencies will be readily apparent because the true path will have much higher anomaly score. We call this approach as *behavioural execution partitioning* for alert investigation. In our attack example, since `IExplorer.exe` has only two socket connections from anomalous websites (one of them is a malicious website from which malware was downloaded) while all the other socket connections were to websites common (normal) in the enterprise. Hence, we can get rid of all the common IP connection vertices and partition the execution of `IExplorer.exe` based on its abnormal behaviour.

Fig. 2b shows the dependency graph generated by NODOZE for our motivating example. It concisely captures the minimal causal path between the root cause (initial socket connection to `IExplorer.exe`) the threat alert (`dropper.exe` socket connection to another host), and all other ramifications (`encryptor.exe` encrypting several files). Observe that in Fig. 2a there are two threat alert events annotated by $\mathcal{E}_1$ and $\mathcal{E}_2$ shown with dashed arrows. Looking at these alert events in isolation, they look similar (both make socket connection to important internal hosts). However, when we consider the ancestry and progeny of each these alert events using backward and forward tracing, we can see that the behaviour of each of them is markedly different.

In order to identify if a threat alert is a true attack or a false alarm, NODOZE uses anomaly scores which quantify the "rareness", or transition probability, of relevant events that have happened in the past. For example, the progeny of alert event $\mathcal{E}_1$ *i.e.* `dropper.exe` $\rightarrow$ `y.y.y.y:445` consists of several events that are more rare *i.e.*, have low transition probability. For example, in the progeny of `Spoolsv.exe` (print service), spawning another process that reads/writes several files happened 0 times in the organization earning this behaviour a high anomaly score. Similarly, in the ancestry of $\mathcal{E}_1$, a chain of events in which an executable is downloaded using Internet Explorer and then connects to a large number of hosts in a short period of times is very rare and thus has a high anomaly score. As a result, when we combine the ancestry and progeny behaviours of $\mathcal{E}_1$, we get a high aggregate anomaly score for the alert.

In contrast, when we consider the progeny of alert event $\mathcal{E}_2$ *i.e.* `Powershell` $\rightarrow$ `z.z.z.z:445`, we see a chain of events that are quite common in an enterprise because these behaviours are exhibited by common Linux utilities (e.g. `diff` and `cut`). Moreover, the ancestry of alert event $\mathcal{E}_2$ contains diagnostic events such as `Tasklist` and `Ipconfig` which are regularly performed to check the health of computers in the enterprise. Therefore, the aggregate anomaly score of $\mathcal{E}_2$ will be quite lower than the anomaly score of $\mathcal{E}_2$.

Once NODOZE has assigned an aggregate anomaly score to

the alert event, it extracts the subgraph from the dependency graph that has the highest anomaly score. The dependency graph for true alert $\mathcal{E}_1$ is shown in Fig. 2b. Observe that in Fig. 2a, `Spoolsv.exe` has created many other socket connections (total 130 sockets); however, the NODOZE generated graph has only `encrypt.exe` process since this behaviour was more anomalous than the other events. Similarly, while `IExplorer.exe` received several socket connections, NODOZE only picked rare IP addresses `a.a.a.a` (malicious website from which malware was downloaded) and `b.b.b.b` since these have higher anomaly scores than the other normal socket connections.

IV. THREAT MODEL & ASSUMPTIONS

Our threat model is similar to existing provenance-based systems [56], [26], [54], [49], [63], [24], [51], [36] *i.e.*, the underlying OS; and provenance tracker are in our trusted computing base (TCB). In this work, we consider an attacker whose goal is to exfiltrate sensitive data, manipulate information present on a system, or to move laterally to other hosts on the network. To achieve this goal the attacker may install malware on the targeted system, exploit a running process, or inject a backdoor.

We make the following assumptions about our system. We assume that the attacker cannot manipulate or delete the provenance record *i.e.*, log integrity is maintained all the time. While log integrity is an important goal, it is orthogonal to the aims of this system and can be ensured by using existing secure provenance systems [35], [25]. We also do not consider the attacks performed using implicit flows (side channels) that do not go through the syscall interface and thus cannot be captured by the underlying provenance tracker. Finally, we do not track attacks exploiting kernel vulnerabilities.

The only underlying TDS's feature that NODOZE relies on is threat alerts. We assume that underlying TDS's detection rate is complete *i.e.* threats related to true attacks are always detected. We also assume that there is at least one event that is anomalous in the ancestry or progeny of alert to categorize it as a true attack. We do not consider Mimicry attacks [62] where attacker evades detection using a sequence of events which are normal in an enterprise. While mimicry is an important consideration, it is out of scope for this work because their detection is actually a limitation of the underlying TDS.

V. PROBLEM DEFINITION

In this section, we first introduce several formal definitions which are required to understand NODOZE's anomaly propagation algorithm and then we formulate the problem statement for NODOZE.

A. Definitions

Dependency Event. OS-level system logs refer to two kinds of entities: subjects and objects. Subjects are processes, while objects correspond to files, socket connections, IPC etc. A dependency (causal) event $\mathcal{E}$ is defined as a 3-tuple $< SRC, DST, REL >$ where $SRC \in \{process\}$ entity that initiates the information flow whereas $DST \in \{process, file, socket\}$ entities which receive information flow, while REL represents information flow relationship. The various kinds of dependency event relationships we consider in this work are shown in Table I. For example, in Fig. 2a a dependency event $\mathcal{E}_1$ is represented as <`dropper.exe`, `y.y.y.y:445`, `IP_Write`>.

TABLE I: Dependency Event Relationships

SRC	*DST*	*REL*
Process	Process	Pro_Start; Pro_End
	File	File_Write; File_Read; File_Execute
	Socket	IP_Write; IP_Read

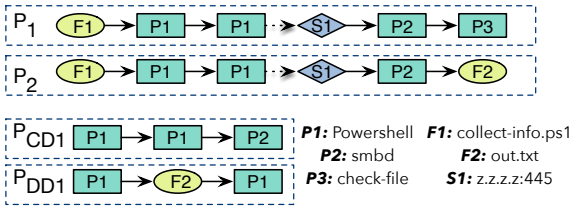


Fig. 4: Example dependency paths of length 5 for alert event $\mathcal{E}_2$ from the motivating example (§II).

Dependency Path. A dependency path P of a dependency event $\mathcal{E}_a$ represents a chain of events that led to $\mathcal{E}_a$ and chain of events induced by $\mathcal{E}_a$. It is an ordered sequence of dependency events and represented as $P := \{\mathcal{E}_1, \mathcal{E}_i, ..., \mathcal{E}_a, ... \mathcal{E}_n\}$ of length n. Each dependency event can have multiple dependency paths where each path represents one possible flow of information through $\mathcal{E}_a$. Dependency path may contain overlapping events, making it possible to represent any dependency graph as a set of dependency paths.

We further divide dependency paths into two categories:

- A control dependency path (CD) of an event ε is a dependency path $P_{CD} = \{\varepsilon_1, \varepsilon_2, ..., \varepsilon_n\}$ such that $\forall\ REL \in \{Pro\_Start, Pro\_End\}$.
- A data dependency path (DD) of an an event ε is a dependency path $P_{DD} = \{\varepsilon_1, \varepsilon_2, ..., \varepsilon_n\}$ such that $\forall\ REL \notin \{Pro\_Start, Pro\_End\}$.

From the motivating attack example, two possible dependency paths $\{P_1, P_2\}$ of length 5, one control dependency path P_{CD1} and one data dependency path P_{DD1} for the alert event $\mathcal{E}_2$ are shown in Fig. 4.

Dependency Graph. All the dependency paths of an event when merged together constitute one single dependency graph. For example, the dependency graph of alert events $\mathcal{E}_1$ and $\mathcal{E}_2$ is shown in Fig. 2a.

True Alert Dependency Graph. As we discussed in §II, due to long running programs there are false dependency events in the dependency graph. Due to false dependencies, there will be unrelated benign events in the dependency graph of a true alert event which might not be causally related to the attack. So we partition the long running programs based on their normal and anomalous behaviour. We call this technique as *behavioural execution partitioning*. This technique will generate a true alert dependency graph, which will contain most anomalous dependency paths. True alert dependency graphs are concise as compared to complete dependency graphs and accelerate

the investigation process without losing vital contextual information about the attack.

B. Problem Statement

Given a list of n alert events $\{\mathcal{E}_1, \mathcal{E}_2, ..., \mathcal{E}_n\}$ and user-specified threshold parameters τ_l and τ_d, we aim to rank these alerts based on their anomaly scores and filter out all the alerts whose anomaly score is less than τ_d as false alarms. Furthermore, we also aim to generate true alert dependency graphs with dependency paths of atmost τ_l length.

There are two key challenges in this problem: 1) assigning anomaly scores to dependency paths of different lengths using historical and contextual information and 2) generating true alert dependency graphs that completely capture attack behaviors. In the next section, we will present a concrete algorithm to assign scores to threat alerts and generate true alert dependency graphs.

VI. Algorithm

In this section, we present a concrete network diffusion algorithm to assign anomaly score to each event in an alert dependency path using historical information, then generate a true alert dependency graph.

A. Roadmap

An anomaly score quantifies the degree of suspiciousness of an event in a dependency path. A naïve way to assign anomaly score is to use frequency of the system events that have happened in the past such that events that are rare in the organization are considered more anomalous. However, sometimes this assumption may not hold since attacks may involve events that happen a lot. From the motivating attack (§II), unzipping a file (`springs.7zip`) is a common event in an organization; however, it was one of the events that led to the attack. Thus, simple frequency-based approach to find anomaly cannot catch such attacks. However, if we consider the chain of events that were informed by `springs.7zip` file, such as initiating a large number of IP connections in a short period of time, we can find out that this is not common behaviour after someone unzips the `springs.7zip` file. Therefore, *our objective is to define the anomaly score not just based on a single event in the dependency path but based on the whole path.* Next, we discuss how to calculate the anomaly scores for each dependency path based on the whole path.

B. Anomaly Score Propagation

In order to calculate a dependency path's anomaly score, we first need to find dependency paths of an alert event. Given a complete dependency graph G of an alert event $\mathcal{E}_\alpha$, we find all the dependency paths of length τ_l for the $\mathcal{E}_\alpha$. To do so, we run depth-first traversal in a backward and forward fashion from the alert event and then we combine those backward and forward paths to generate unified paths such that each unified path contains both the ancestry and progeny causal events of alert. In Algorithm 1, Lines 2 to Lines 6 show the dependency path search algorithm. Function GetDependencyGraph generates a complete dependency graph of an input event, functions GetSrcVertex and GetDstVertex return SRC and DST entities of input event respectively, functions DFSTraversalBackward and DFSTraversalForward return backward and forward dependency paths for input event respectively, and function CombinePaths combine backward and forward paths.

After generation of dependency paths for candidate alert event, NoDoze assigns anomaly scores to each event in the dependency paths. In Algorithm 1, Lines 7 to Lines 10 show this process. To calculate the anomaly scores, we first construct a $N \times N$ transition probability matrix M for the given dependency graph G of alert event, where N is the total number of vertices in G. Each matrix entry M_ε is computed by the following equation:

$$M_\varepsilon = probability(\varepsilon) = \frac{|Freq(\varepsilon)|}{|Freq_{src\_rel}(\varepsilon)|} \tag{1}$$

Here, $Freq(\varepsilon)$ represents how many times the causal event ε has happened in the historic time window with all 3-tuple of ε exactly same, while $Freq_{src\_rel}(\varepsilon)$ represents how many times event ε where only SRC and REL from 3-tuple are exactly same. Hence, M_ε means the happening probability of this specific event. If ε event never happened before in historical information, then its value is 0. On the other hand, if ε is the only event between SRC and any other entity with REL in our historical information then its value 1. Note that this anomaly score assignment algorithm is an unsupervised algorithm with no training phase. To count the frequency of events that have happened in the past we built an Event Frequency Database that periodically stores and updates events frequency in the whole enterprise. A detailed discussion regarding the construction of such database will be provided in §VII.

Let's consider an alert event $\mathcal{E}_1$:=<`dropper.exe`, `y.y.y.y:445`, `IP_Write`> from Fig. 2a. We first calculate $Freq(\mathcal{E}_1)$ by counting the number of events that have happened in our frequency event database where $SRC \in$ `dropper.exe` , $DST \in$ `y.y.y.y:445` and REL is `IP_Write`. Then, we will calculate $Freq_{src\_rel}(\mathcal{E}_1)$ by counting the number of events where $SRC \in$ `dropper.exe` and REL is `IP_Write` while DST could be any entity node. Details regarding how these functions are implemented will be provided in §VII.

Transition probability for a given event tells us the frequency with which a particular source flows to a particular destination; however, we are ultimately going to propagate this value through the graph, but when we do so we want to account for the total amount of data flowing out of the source, and the total amount of data flowing into the destination. For this, we calculate IN and OUT score vectors for each entity in the dependency graph G. The IN and OUT scores represent the importance of an entity as an information receiver and sender respectively. In other words, IN and OUT scores measure the degree of fanout in either direction for each entity in the graph. For example, in the motivating attack (§II), the `IExplorer.exe` process entity has both high IN and OUT scores, as it frequently reads and writes to socket connections. On the other hand, `dropper.exe` process entity has a high OUT score as it frequently writes to socket connections but has low IN since it does not read anything. We provide a detailed algorithm to calculate these vectors in §VI-C.

Algorithm 1: GETPATHANOMALYSCORE

Inputs : Alert Event $\mathcal{E}_\alpha$;
Max Path Length Threshold τ_l
Output: List $L_{<P,AS>}$ of dependency path and score pairs.

G_α = GETDEPENDENCYGRAPH($\mathcal{E}_\alpha$)
$V_{src} \leftarrow$ GETSRCVERTEX($\mathcal{E}_\alpha$)
$V_{dst} \leftarrow$ GETDSTVERTEX($\mathcal{E}_\alpha$)
$L_b \leftarrow$ DFSTRAVERSALBACKWARD(G_α,V_{src},τ_l)
$L_f \leftarrow$ DFSTRAVERSALFORWARD(G_α,V_{dst},τ_l)
/* *Combine Backward and Forward Dependency Paths* */
$L_p \leftarrow$ COMBINEPATHS(L_b,L_f)
/* *Generate a transition matrix of an input graph using Eq. 1* */
M = GETTRANSITIONMATRIX(G)
foreach $P \in L_p$ **do**
  /* *Calculate Path anomaly score using Eq. 2 and Eq. 3* */
  $AS \leftarrow$ CALCULATESCORE(P,M)
  /* *Append path and its anomaly score to a list* */
  $L_{<P,AS>} \leftarrow L_{<P,AS>} \cup < P, AS >$
end
return $L_{<P,AS>}$

Once the transition probability matrix and IN and OUT scores calculation are done, we calculate the regularity (normal) score of each dependency path. Given a dependency path $P = (\varepsilon_1, ..., \varepsilon_l)$ of length l, the regularity score $RS(P)$ is calculated as follows:

$$RS(P) = \prod_{i=1}^{l} IN(SRC_i) \times M(\varepsilon_i) \times OUT(DST_i) \quad (2)$$

where IN and OUT are the sender and receiver vectors, and M is calculated by Equation 1. In Equation 2, $IN(SRC_i) \times M(\varepsilon_i) \times OUT(DST_i)$ measures the regularity of the event ε that SRC_i sends information to DST_i entities. After calculating regularity score, we calculate the anomaly score as follows:

$$AS(P) = 1 - RS(P) \quad (3)$$

According to this equation, if any path that involves at least one abnormal event, it will be assigned a high anomaly score as it will be propagated to the final score. In Algorithm 1, function CALCULATESCORE generates anomaly scores of given dependency paths.

C. IN and OUT Scores Calculation

As mentioned above, Equation 2 requires the IN and OUT score vectors for each entity in the dependency graph. We populate IN and OUT score for each entity, based on its type as follows:

Process Entity Type. To assign IN and OUT score to a candidate process entity we check the historical behaviour of candidate process entity globally in the enterprise and calculate its scores as follows: Let v be the candidate process entity in the dependency graph and m is a fixed time window length. The period from the time v is added to the dependency graph (T_0) to the current timestamp (T_n) is partitioned into a sequence of time windows $T = \{T_0, T_1, ..., T_n\}$, where T_i is a time window of length m. If there is no new edge from/to vertex v in window T_i, then T_i is defined as a *stable window*. The vertex v's IN and OUT score is calculated using Equation 4 and Equation 5 respectively where $|T'_{from}|$ is the count of stable windows in which no edge connects from v, $|T'_{to}|$ is the count of stable windows in which no edge connects to v, and $|T|$ is the total number of windows.

$$IN(v) = \frac{|T'_{to}|}{|T|} \quad (4) \qquad OUT(v) = \frac{|T'_{from}|}{|T|} \quad (5)$$

To understand the intuition of these equations, consider an example where a process vertex constantly have new edges going out from it while there is no edge going in. In such a case, the vertex has very low IN score, its OUT score will be high. If there is suddenly an edge going in the vertex, it is abnormal. The range of process entity IN and OUT score $\in [0, 1]$, when a node has no stable window, *i.e.*, the node always has new edges in every window, its score is 0. If all the windows are stable, the node stability is 1. Through repeated experimentation, we typically set the window length 24 hours. Hence the stability of a node is determined by the days that the node has no new edges and the total number of days.

Data Entities. Data entity type consists of file and socket entities. Data entities cannot be assigned global scores like Process entity as mentioned-above because the behaviour of data entity various from host to host in the enterprise. We define local values in terms of low and high IN and OUT scores for data entities. To assign IN and OUT scores for file entity vertices, we divide the file entities into three types and based on the type, we assign IN and OUT scores. *1) Temporary Files:* All the file entities which are only written and never read in the dependency graph are considered as temporary files as suggested by [47]. We give temporary files as high IN and OUT scores since they usually do not contribute much in attack anomaly score. *2) Executable Files:* Files which are executable (execute bit is 1) are given low IN and OUT since they are usually used in the attack vector thus important sender and receiver of information. *3) Known malicious extensions:* We use an online database [9] of known malicious file extensions to assign low IN and OUT to such files since they are highly anomalous. All the other files are given IN and OUT score of 0.5. To assign IN and OUT scores for socket connection entities, we use domain-knowledge. We use an online database of malicious IP [10] address to assign low IN and OUT score.

D. Anomaly Score Normalization

For each alert causal path P, we calculate the anomaly score using Eq. 2 and Eq. 3. However, it is easy to see that longer paths would tend to have higher anomaly scores than the shorter paths. To eliminate the scoring bias from the path length, we normalize the anomaly scores so that the scores of paths of different lengths have the same distribution.

We use a sampling-based approach to find the decay factor which will progressively decrease the score in Equation 2. To calculate decay factor α, we first take a large sample of false alert events. Then, for each alert we generate the dependency paths of different max lengths τ_l and generate anomaly score for those paths. Then we generate a map M which contains average anomaly scores for each path length. Using this map, we calculate the ratio at which the score increases with increasing length from the baseline length k and use this ratio decay factor α. The complete algorithm to calculate the decay factor α using the sampling method is

Algorithm 2: CALCULATEDECAYFACTOR

Inputs : List of false alert causal events $L_{\mathcal{E}}$;
Baseline length k;
Max. Path Length Threshold τ_l
Output: Decay Factor α

M = KeyValue Store of Path Length and Avg. Anomaly Score
foreach $\mathcal{E} \in L_{\mathcal{E}}$ **do**
  for $i \leftarrow 0$ **to** τ_l **do**
    /* Use Algorithm 1 to generate anomaly score for given event and max path length */
    $L_{<P,AS>}$ = GETPATHANOMALYSCORE($\mathcal{E}$,i)
    /* Takes the average of anomaly scores for each path length and store in map */
    $M[i] \leftarrow$ AVERAGESCORE($L_{<P,AS>}$,$M[i]$)
  end
end
/* Returns the ratio at which score increases with length from the baseline */
$\alpha \leftarrow$ GETDECAYFROMBASELINE(M,k)
return α

Algorithm 3: DEPENDENCY PATHS MERGE

Inputs : L_{PS} List of dependency path P and score S pairs;
Merge Threshold τ_m
Output: Alert Dependency Graph G

/* Sort list by anomaly scores */
L_{PS} = SORTBYSCORE(L_{PS})
for $i \leftarrow 0$ **to** SIZEOF(L_{PS}) $- 1$ **do**
  /* Path and its anomaly score pair */
  $< P_1, S_1 > \leftarrow L_{PS}[i]$
  $< P_2, S_2 > \leftarrow L_{PS}[i+1]$
  if $S_1 - S_2 < \tau_m$ **then**
    $G \leftarrow G \cup P_1$
    $G \leftarrow G \cup P_2$
  end
end
return G

shown in Algorithm 2. Once the decay factor is calculated, the regularity score Equation 2 becomes as follows:

$$RS(P) = \prod_{i=1}^{l} IN(SRC_i) \times M(\varepsilon_i) \times OUT(DST_i) \times \alpha \tag{6}$$

This equation returns a normalized anomaly score for a given dependency path P of length l.

E. Paths Merge

As attacks are usually performed in multiple steps, it is not possible to capture the complete causality of a true alert event by returning the single dependency path that is most anomalous. Likewise, returning the full dependency graph (comprised of all paths) to cyber analysts is inaccurate because it contains both anomalous paths as well as benign paths that are unrelated to the true alert. To strike a balance between these two extremes, we introduce a merging step that attempts to build an accurate true alert dependency graph by including only dependency paths with high anomaly scores.

A naïve approach to this problem would be to return the top *k* paths when ranked by anomaly score; this solution is not acceptable because not all attacks contain the same number of steps, which could lead to the admission of benign paths or the exclusion of truly anomalous paths. Instead, we present an algorithm that uses a best effort approach to merge paths together in order to create an optimally anomalous subgraph. Through experimentation with NODOZE, we found that there is an orders of magnitude difference between the scores of benign paths and truly anomalous paths. Because of this, we are able to introduce a merge threshold τ_m which quantifies the difference between the two. Algorithm 3 shows how to merge dependency paths based on the merge threshold τ_m. At a high level, this algorithm keeps merging high anomaly score paths until the difference is greater than τ_m. In order to calculate an acceptable value for τ_m, we use a training phase to calculate the average difference between anomalous and benign paths. While the availability of labeled training data that features true attacks may seem prohibitive, recall that NODOZE is designed for enterprise environments that already employ trained cyber analysts; thus, the availability of training data is a natural artifact of their work. We also note that, based on our experience, the τ_m threshold only needs to be calculated once per deployment.

F. Decision

The main goal of NODOZE is to rank all the alerts in a given timeline. However, we can also calculate a decision or a cut-off threshold τ_d, which can be used to decide if a candidate threat alert is a true attack or a false alarm with high confidence. If anomaly score of a threat alert is greater than the decision threshold than it is categorized as a true alert otherwise a false alarm. To this end, calculating τ_d require training dataset with true attacks and false alarms and its value depends on the current enterprise configuration such as the number of hosts and system monitoring events.

G. Time Complexity of our Algorithm

The dependency paths search for an alert event is done with D depth-bounded Depth-first search traversal. We execute DFS twice for each alert, once forward and once backward to generate both forward tracing and backward tracing dependency paths. So time complexity is $\mathcal{O}(|b^D|)$ where b is the branching factor of the input dependency graph. Equation 2 runs for each path so time complexity is $\mathcal{O}(|PD|)$ where P is the total number of dependency paths for the alert event.

VII. IMPLEMENTATION

We implement NODOZE for an enterprise environment. We collected system event logs in PostgreSQL database using Windows ETW [1] and Linux Auditd [11]. Our implementation consists of 3 major modules: a) Event Frequency Database Generator, b) Alert Triage & Graph Generator, and c) Visualization Module.

A. Event Frequency Database

In order to calculate the transition probability matrix M, IN score vector, and OUT score vector for Equation 2, we implemented Event Frequency Database in 4K lines of Java code. For a given a time period, this module counts the number of events that have happened in an enterprise network, then stores these counts in an external database. During runtime, NODOZE queries this database to calculate event frequencies.

Users of NODOZE can periodically run this module to update the enterprise-wide event frequencies. To remove non-deterministic and instance specific information in each event's SRC and DST entities such as timestamp and process id, we abstract/remove such fields before storing these events. Our abstraction rules for each of the entity types are similar to previous works [36], [49] with some changes to fit our analysis:

– *Process Entity*. We remove all the information in the process entities except the process path, commandline arguments and `gid` (group identification number).
– *File Entity*. We remove the inode and timestamps fields from the file entities while abstract file paths by removing user specific details. For example, `/home/user/mediaplayer` will be changed to `/home/*/mediaplayer`.
– *Socket Entity*. Each socket connection entity has two addresses i.e. source ip and destination ip each with port number. connection is outgoing we remove the source IP and its port which is chosen randomly by the machine when initiating the connection. If the connection is incoming we the remove destination IP and its port. The end result is that external IP of the connection is preserved while the internal address is abstracted.

The final equations to calculate the frequencies of an event $\mathcal{E}_i =< SRC_i, DST_i, REL_i >$ which are used in transition probability matrix generation (Eq. 1) are as follows:

$$Freq(\mathcal{E}_i) = \sum_{h}^{hosts} checkEvent(SRC_i, DST_i, REL_i, h, t) \quad (7)$$

$$Freq_{src\_rel}(\mathcal{E}_i) = \sum_{h}^{hosts} checkEvent(SRC_i, *, REL_i, h, t) \quad (8)$$

where $hosts$ are hosts in the enterprise environment while $checkEvent$ function returns the number of times event $\mathcal{E}_i$ has occurred on the $host$. We only count event $\mathcal{E}_i$ once in time window t for a host to prevent poisoning attacks [42]. Note that in our experiments t is set to stable window size (discussed in §VI-C) which is 1 day. Finally, in Eq. 8 "$*$" means any DST entity.

B. Alert Triage and Graph Generation

We implemented NODOZE's network diffusion algorithm and concise alert dependency graph generation in 9K lines of Java code. We also implemented several optimizations such as an event frequency cache to minimize the NODOZE overhead.

We implement a basic dependency graph generator that, given an event parses the audit logs from Linux and Windows stored in PostgreSQL and generates the dependency graph on-the-fly. We also introduced several summarization techniques that make a graph more suitable for NODOZE analysis without affecting the correctness of causality analysis:

– *Merge Transient Processes*. There are processes in the provenance graph whose sole purpose is to create another process. We merge such processes into one node since this does not affect our analysis. Consider the dependency graph in Fig. 2a, `IExplorer.exe` process entity spawns another `IExplorer.exe` process entity. We merge these two `IExplorer.exe` process entities together.
– *Merge Similar Sockets Connection*. Socket connections going out to same address from the same process vertex have multiple vertices in the raw dependency graphs. We merge such socket connections into a single vertex. From the perspective of alert event causality analysis, this does not affect correctness but saves NODOZE's time during dependency path generation.

C. Visualization Module

We have built a front-end which helps cyber analysts to visualize NODOZE's concise dependency graphs. We use GraphViz [33] to generate causal graph in a `dot` format and then convert the dot file into html format. Cyber analysts can use these html-based graphs to visualize the most anomalous dependency paths with their anomaly scores.

VIII. EVALUATION

In this section, we focus on evaluating the efficacy of NODOZE as an automatic threat alert triage and investigation system in an enterprise setting. In particular, we investigated the following research questions (RQs):

RQ1 How accurate is NODOZE over existing TDS? (§VIII-C)
RQ2 How much can NODOZE reduce the dependency graph of a true alert without sacrificing the vital information needed for investigation? (§VIII-D)
RQ3 How much of investigator's time can NODOZE save when used in an enterprise setting? (§VIII-E)
RQ4 What is the runtime overhead of NODOZE? (§VIII-F)

A. Experiment Setup

We monitored and collected OS-level system events and threat alerts at NEC Labs America. In total, we monitored 191 hosts (51 Linux and 140 Windows OS) for 5 days which were used daily for product development, research and administration at NEC Labs America. During this time span, we also simulated 50 attacks which include 10 real-world APT attacks and 40 recent malwares downloaded from VirusTotal [19]. A short description of each APT attack with generated threat alert is shown in Table II.

We deployed NODOZE on a server with Intel®Core(TM) i7-6700 CPU @ 3.40GHz and 32 GB memory running Ubuntu 16.04 OS. We used the baseline TDS [8] to generate threat alerts. In summary, our experiment contains 400 GB of system monitoring data with around 1 billion OS-level log events and 364 threat alerts. The Event Frequency Database in our experiments was populated using 10 days of OS-level system events. Note that our evaluation dataset of 364 labeled alert scenarios was generated after the event frequency database was populated.

B. Baseline TDS

The baseline TDS we used to generate threat event alerts is a commercial tool [8]. Details regarding anomaly detection models used in this tool can be found here [31]. At a very high level, this TDS applies an embedding based technique to detect anomalies. It first embeds security events as vectors. Then, it models the likelihood of each event based on the embedding vectors. Finally, it detects the events with low likelihood as anomalies.

TABLE II: Real-world attack scenarios with short descriptions and generated threat alerts by underlying TDS.

Attacks	Short Description	True Threat Alert
WannaCry [18]	Motivating example discussed in §II	See §II
Phishing Email [16]	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	<Excel.exe, java.exe, Pro_Start>
Data Theft [49]	An attacker downloaded a malicious bash script on the data server and used it to exfiltrate all the confidential documents on the server.	<ftp, y.y.y.y:21, IP_Write>
ShellShock [13]	An attacker utilized an Apache server to trigger the Shellshock vulnerability in Bash multiple times.	<bash, nc.traditional, Pro_Start>
Netcat Backdoor [17]	An attack downloaded the netcat utility and used it to open a Backdoor, from which a Persistent Netcat port scanner was then downloaded and executed using PowerShell	<nc.exe, cmd.exe, Pro_Start>
Cheating Student [51]	A student downloaded midterm scores from Apache and uploaded a modified version.	<Apache2, /www/newscores, File_Write>
Passing the Hash [7]	An attack connected to Windows domain Controller using PsExec and run credential dumper (*e.g.*, gsecdump.exe).	<gsecdump.exe, g64-v2b5.exe, Pro_Start>
wget-gcc [67]	Malicious source files were downloaded and then compiled.	<wget, x.x.x.x:80, IP_Read>
passwd-gzip-scp [67]	An attack stole user account information from passwd file, compressed it using gzip and transferred the data to a remote machine	<scp, x.x.x.x:22, IP_Write>
VPNFilter [20]	An attacker used known vulnerabilities [13] to penetrate into an IoT device and overwrite system files for persistence. It then connected to outside to connect to C2 host and download attack modules.	</var/vpnfiler, x.x.x.x:80, IP_Read>

C. Improvement Over Existing TDS

The first research question of our evaluation is how much NoDoze improves the accuracy of existing TDS [31], [43], [29], [60] which are based on heuristics and single event matching rules. To answer this question, we used NoDoze along with the baseline TDS [8]. In our experiment, we used the baseline TDS to monitor the system activities of the enterprise for anomalies and generate threat alerts. We then manually labeled these alerts as true positives and false positives and use them as the ground truth to evaluate NoDoze. Lastly, we used NoDoze to automatically label the alerts and compared the results with the ground truth.

In our experiments, the baseline TDS generated a total of 364 alerts (50 true alerts and 314 false alarms). The detection accuracy of NoDoze is measured using true positive rate (TPR) and false positive rate (FPR). Intuitively, the FPR measures the total number of false alerts that were categorized as true attacks by NoDoze. By adjusting the decision threshold τ_d, NoDoze can achieve different TPR and FPR as shown in the ROC graph in Fig. 5. When the threshold is set to detect 100% of true positives, NoDoze has a 16% FPR. In other words, NoDoze can reduce the number of false alerts of the baseline TDS by more than 84% while maintaining the same capability to detect real attacks. Fig. 6 shows the cumulative distribution function for ranked true and false alerts based on aggregate anomaly scores. The decision threshold (shown with red line), when set to 100% of true positives, removes the large portion of false alerts because the true positives are substantially ranked higher than false alerts.

D. Accuracy of Capturing Attack Scenarios

To answer RQ2, we used NoDoze to capture the attack scenarios of 10 APT attacks from their complex provenance graphs. We evaluate NoDoze on the APT attacks because we know the precise ground truth dependency graphs of the attacks. The results are summarized in Table III. The duration columns represent the time taken in seconds by underlying provenance tracker to generate a complete dependency graph and time taken by NoDoze to run its analysis and generate a concise graph.

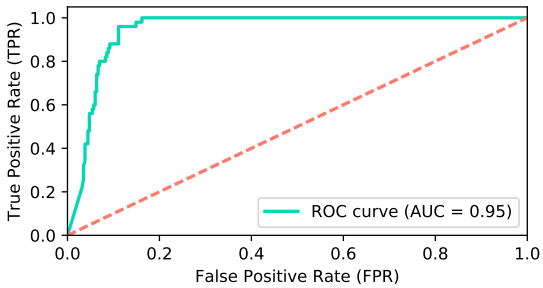


Fig. 5: ROC curve for our experiments using NoDoze along with TDS.

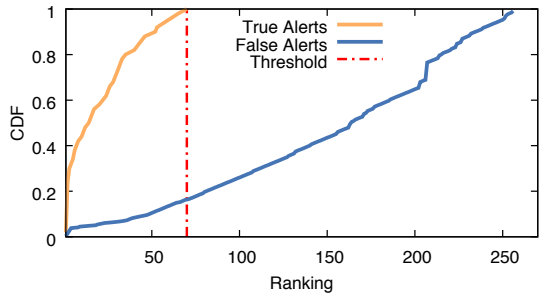


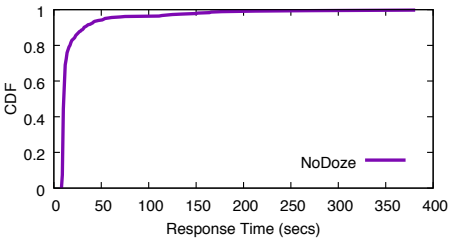
Fig. 6: CDF for ranked true and false alerts based on aggregate anomaly score.

Our experiment shows that our system accurately extracts the APT attack scenarios from the complex provenance graphs generated by the underlying provenance tracker. NoDoze can reduce the size of the provenance graph by two orders of magnitude. Such a reduction may substantially reduce the work load of cyber analyst when investigating the threat alerts and planning incident responses.

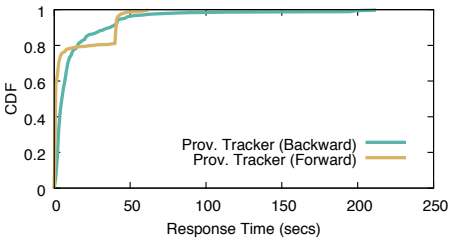
We also measured the completeness of the NoDoze generated dependency graph for each attack. We measured completeness in terms of two metrics: control dependency (CD) and data dependency (DD) (discussed in §V) with their true positive (TP) and false positive (FP) rates. Intuitively, the TP means the number of truly attack related edges present

TABLE III: Comparison of dependency graphs generated by NODOZE against prior tools [26], [41]. Completeness means how much our graph able to capture the attack dependency graph in terms of CD and DD with their true positive (TP) and false positive (FP) rates.

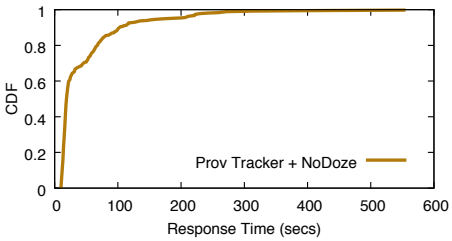
Attacks	Baseline Prov. Tracker				NODOZE				NODOZE Completeness			
	Dur.(s)	#Ver.	#Edg.	Size(KB)	Dur.(s)	#Ver.	#Edg.	Size(KB)	CD(TP)	CF(FP)	DD(TP)	DD(FP)
WannaCry	94.0	5948	8712	3,320	18.0	19	21	49	100%	0%	100%	0.03%
Phishing Email	63.0	2095	6002	3,984	10.0	17	16	48	100%	0%	100%	0%
Data Theft	73.0	5364	23825	2,208	41.0	23	24	65	100%	0%	100%	0%
ShellShock	31.0	2794	4031	3,776	8.0	15	20	36	100%	0%	100%	0%
Netcat Backdoor	62.0	2914	6158	1,968	14.0	12	11	48	88%	0%	84%	0%
Cheating Student	50.0	1217	22647	784	10.0	12	11	40	100%	0%	100%	0.07%
Passing the Hash	53.0	848	1026	560	11.0	8	8	36	100%	0%	90%	0%
wget-gcc	63.0	8323	8679	168	9.0	11	12	33	100%	0%	100%	0.01%
passwd-gzip-scp	68.0	8066	15318	5,168	8.0	10	9	36	100%	0%	100%	0%
VPNFilter	20.0	2639	9774	1,000	9.0	15	15	45	100%	0%	100%	0%



(a) NODOZE response times to assign scores and build concise graphs.



(b) Underlying provenance tracker response times to build complete graphs.



(c) End-to-end response times from input alerts to the results.

Fig. 7: CDF of query response times for all the 364 threat alerts in our dataset.

in the concise graph generated by NODOZE. For all 10 APT attacks, we were able to recover the alert's expected control dependency graph except for Netcat attack where the expected length of control dependency path was larger than user-defined τ_l. Note, this does not affect the correctness of causality analysis since cyber analysts can increase the depth of the path by increasing τ_l during the alert investigation. In some cases, we were not able to completely recover the data dependency graph because incorporating those data dependencies required larger merge threshold τ_m than set in our experiments. However, increasing the merge threshold also increases the number of FP in the data dependencies. Thus, finding the best possible thresholds which strike a balance between TP and FP require training run once before deployment in an enterprise.

Nevertheless, NODOZE decreases the size of original graph by *two orders of magnitude* which accelerates the alert investigation without losing vital information about attack. To further explain how well can NODOZE capture the attack scenarios, we will discuss two attack cases from Table II in §IX.

E. Time Saved Using NODOZE

Recent studies [2], [6] have shown that it takes around 10-40 mins to manually investigate a single threat alert in an enterprise. This time spent on an investigation is also known as Mean-Time-To-Know in industry. Note that these studies have also confirmed that cyber analysts receive around 60-80% false alarms using existing TDS, which was also the false alarm rate of the baseline TDS used in our experiments.

If we conservatively assume cyber analysts spend 20 mins on average on each false alarm in our experiments they would have to waste around 104 employee-hours on investigating those false alarms. However, NODOZE reduces false alarms of existing TDS by 84%, which saves around 90 employee-hours in an enterprise setting.

F. Runtime Performance of NODOZE

To answer RQ4, we measured the runtime overhead of NODOZE for all the alerts in our dataset. NODOZE's response time for all the 364 alerts events is shown as a CDF in Fig. 7a. This response time includes running anomaly propagation algorithm and generating a concise dependency graph for given alerts. Results show that 95% of all the alerts are responded by NODOZE in less than 40 seconds. There are few cases where NODOZE took a long time to respond. In these cases, most time was spent on constructing a large transition probability matrix for a large input dependency graph.

To further understand why NODOZE has large response times in some cases, we also measured the dependency graph generation query response times for all 364 alerts in our dataset. The results are shown in Fig. 7b. Complete graph generation also has long response times because of extra large dependency graph construction. For these large dependency graphs, NODOZE also incurs larger overhead due to the reasons mentioned above. However, because we rarely encountered this issue in our experiments and other provenance tracking techniques [49], [52] also suffer from this performance problem, we leave solving this problem for future work. CDF for end-to-end response time starting from the time alert is received until the alert is triaged is shown in Fig. 7c. 95% of the threat alerts are responded in less than 200 seconds. Note that right now NODOZE analysis framework runs on a single machine using single thread; however, NODOZE can be

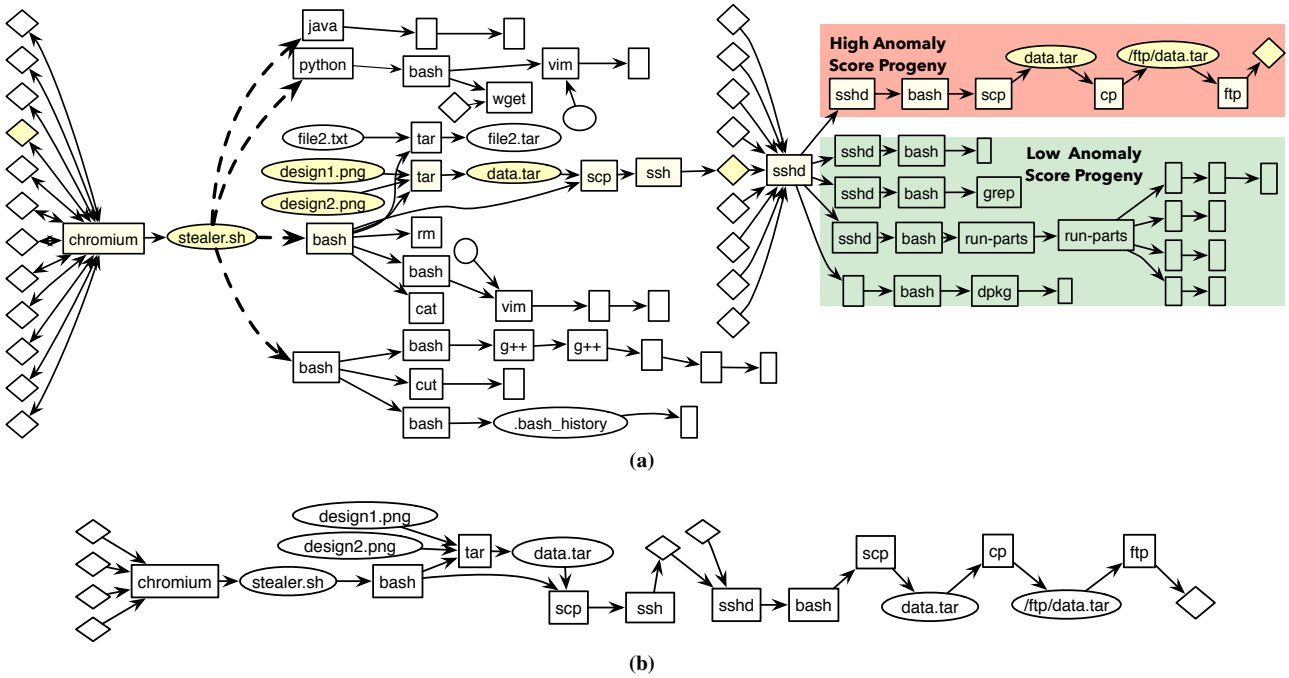


Fig. 8: Data theft attack scenario discussed in §IX-A. **(a)** Part of dependency graph generated by traditional tools. **(b)** Concise true alert dependency graph generated by NoDoze.

parallelized easily using existing distributed graph processing frameworks (details in §X).

IX. Case Studies

A. Data Theft Attack

Scenario. In this attack, an employee of a mobile app development company stoles app designs that were about to be released and posted them online. To perform this data theft attack, employee downloads a malicious bash script (`stealer.sh`) to the data server via `HTTP`. Bash script (`stealer.sh`) discovers and collects all the application designs on the server. Then, the script compresses (`tar`) all the design files (`design1.png` and `design2.png`) into a single tarball, transferred the tarball to a low-profile desktop computer via `SSH`, and finally uploads it to an external `FTP` server under employee's control. Since the employee is aware of the company's TDS, the bash script also creates a bunch of spurious processes to create false alerts which buys the employee enough time to complete the attack and post the designs online.

Threat Alerts. Once the bash script is executed many threat alerts are generated by TDS in a short period of time, which are investigated by cyber analyst one by one. The dependency graph of these threat alert is shown in Fig. 8a where dashed edges show threat alert events.

Alert Investigation. Without NoDoze, an investigator will generate a complete dependency graph for each of the threat alerts generated by TDS and manually inspect them only to see that just 1 of the 4 threat alerts was true attack. However, by the time investigator has examined all the false alerts (∼1.6 hours), all the app designs may have already been posted online.

On the other hand, NoDoze will ingest all these threat alerts and rank them based on their anomaly score. In this scenario, all the false alerts intentionally created by the attacker will be ranked lower while the true alert will be ranked higher. The threat alert events which led to data theft will be ranked on the top because of various rare events in its progeny. For example, using `cp` utility to copy `data.tar` to the ftp serving directory and using `ftp` to make a connection outside the organization. This chain of events has never happened in the organization. Contrary to this, events in the progeny of all the false threat alerts were quite common such as running `g++` and Linux utilities.

NoDoze also generates a concise dependency graph with only data theft dependency paths, while all the benign paths have been removed as shown in Fig. 8b. Observe that in Fig. 8a, the progeny of the true threat alert event has various operations such as `run-parts` and `dpkg`, which are removed in the NoDoze generated graph because their anomaly score is lower than the data theft dependency path. Note that NoDoze generated graph has some socket vertices connected to `chromium` and `sshd` which are unrelated to attack but they are included in NoDoze's graph because they were rare. Note that Table III shows the FP rate higher than 0 due to these unrelated socket connections. Nevertheless, NoDoze decreases the original graph size by 2 orders of magnitude.

B. ShellShock Attack

Scenario. In this attack scenario, attacker targets a ShellShock vulnerable Apache webserver to open several reverse shells and steals sensitive files. The attacker launches the attack in two phases. In the first phase, the attacker runs some Linux utilities (*e.g.*,`ls`, `top`) without doing serious damage. In the second phase, the attacker tries to discover sensitive data on webserver using commands such as '`/bin/cat /etc/passwd`' and '`/bin/cat /var/log/access_logs`' . Once the sensitive files are found, the

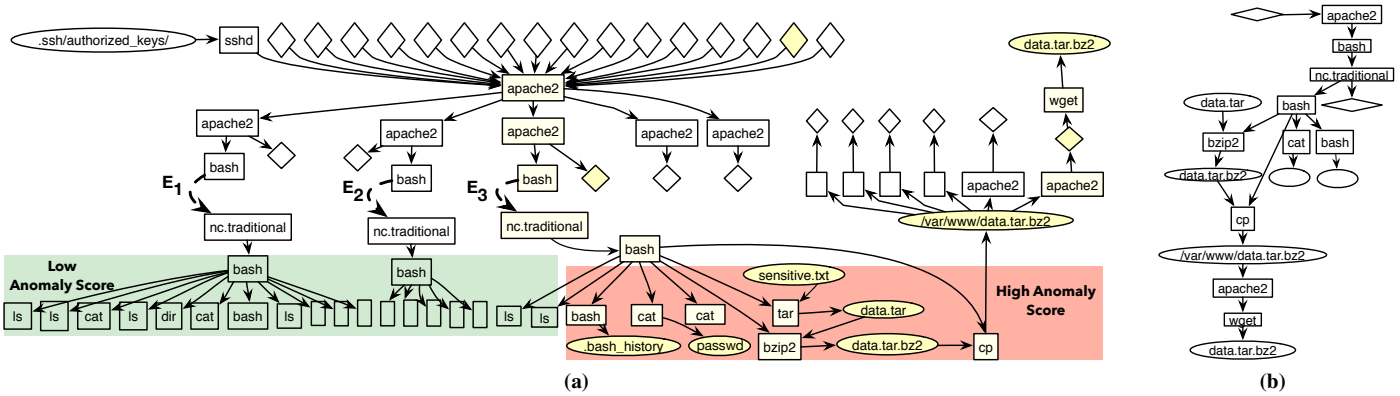


Fig. 9: ShellShock attack scenario discussed in §IX-B. **(a)** Part of dependency graph generated by traditional tools. **(b)** Concise true alert dependency graph generated by NODOZE.

attacker archived (tar) and compressed (bzip2) the sensitive files and transferred (`cp`) it to Apache hosting directory so that attacker can download (`wget`) it from another machine. Once this phase is done, the attacker erased the history of bash commands by removing `.bash_history`. Later, noises were introduced when a normal user opened new bash terminals. These terminals read the modified `.bash_history` creating a false causal link to the attack.

Threat Alerts. This attack scenario simulation generated various threat alerts events because spawning a `nc.traditional` from `bash` process is considered as anomalous behaviour by TDS. These threat alert events are indicated by dashed arrows in Fig. 9a.

Alert Investigation. The forward dependency graph of all the threat alert events consists of `bash` commands which are quite common in the enterprise such as `ls` and `cat`. However, alert event $\mathcal{E}_3$ consist of other commands which are not common in an organization such as '`/bin/cat /etc/passwd`' and '`cp data.tar.bz2 /var/www/`'. All the threat alert events will be ranked lower than alert event $\mathcal{E}_3$ because the progeny graph of alert event $\mathcal{E}_3$ contains most anomalous dependency paths as compared to other alert events.

Fig. 9b shows the concise dependency graph generated by NODOZE. NODOZE's graph only has data exfiltration while all the common terminal commands are excluded from this graph because of our behavioural execution partitioning technique. This technique chooses alert dependency paths that have data exfiltration events over other dependency paths that launched common terminal commands due to two reasons: 1) creation and transfer of new files have low frequency in our event database since these do not happen very often as compared to running Linux utilities; 2) the dependency path for data exfiltration also `wget`'s sensitive files on other machines while benign paths do not include any more anomalous behaviour.

X. LIMITATIONS & DISCUSSIONS

We outline the limitations of NODOZE through a series of questions. We also discuss how NODOZE can be extended under different scenarios.

What happens if an attacker uses benign process and file names for an attack? NODOZE is resilient to changes in the file and process names. At first glance, it may seem surprising; however, NODOZE inherits this from the use of data provenance, which captures true causality, not merely correlations. Even if the attacker starts a malware with a benign program name such as `Notepad`, the causality of the program such as how it was spawned and what changes it induced differentiates its behaviour from the normal behaviour of `Notepad`. Note that this property sets our work apart from heuristics-based TDS (*e.g.*, [68], [32]).

Can NODOZE *be extended to incorporate distributed graph processing frameworks for improved performance?* NODOZE uses a novel network diffusion algorithm to propagate the anomaly scores on the edges of a large dependency graph to generate an aggregate anomaly score. One can potentially parallelize this algorithm using existing large-scale vertex-centric graph processing frameworks [53]. In this work, we do not enable distributed graph processing; however, we will explore this option in future work.

Can NODOZE *run anomaly propagation algorithm while generating the dependency graph from audit logs?* Currently, NODOZE first generates a complete dependency graph and then it propagates the anomaly score on that dependency graph. However, one can design a framework which propagates the anomaly score while generating the large dependency graph using iterative deepening depth first search and stop the analysis if anomaly scores do not increase in next iteration. In this way additional step of generating a large dependency graph first can be removed completely.

What is the role of underlying TDS in NODOZE*'s effectiveness?* NODOZE is essentially an add-on to existing TDS for false alarm reduction. Thus, NODOZE can detect true attack only if it was detected by the underlying TDS first. If underlying TDS misses true attack and does not generate an alert, then NODOZE will not be helpful. Improving the true detection rate of underlying TDS is orthogonal to our work; however, our findings suggest that path-based context could be a powerful new primitive in the design of new TDS.

Does the choice of underlying TDS affect the accuracy of NODOZE*?* NODOZE is independent to the choice of underlying TDS, we used this TDS [8] in particular because it was licensed by our enterprise; licensing additional TDS for our

evaluation, which was conducted on hundreds of hosts, was prohibitively costly. However, this tool is a state-of-the-art commercial TDS that is based on a reputable peer-reviewed detection algorithm [31] and is similar to existing commercial and academic TDS [3], [5], [43], [29], [21] in FPR.

XI. Related work

In §II, we described the limitations of combining TDS with provenance tracker that NoDoze addresses, and complement the discussion on related work here.

Threat Detection. Existing threat detection approaches can be classified as online and offline approaches. Online detection approaches often look for a specific sequence of syscalls to detect malicious programs in a running system [37], [48]. While offline approaches leverage forensic analysis on audit log to find the root cause of intrusion. Due to performance and space constraints, online approaches do not keep audit logs to support forensic analysis. On the other hand, existing offline approaches are labor intensive which makes them prohibitively impractical in an enterprise.

To improve syscall based methods, Tandon *et al*. [61] considered syscall arguments in addition to syscall sequences for malicious program detection. Sekar *et al*. [57] further added more complex structures such as loops and branches in the syscall sequences. However, all these syscall based systems suffer from a high false alarm rate due to the lack of contextual information. NoDoze use historical contextual information of system activities with more domain information such as process names and commandline arguments to achieve better accuracy. Researchers have also proposed to automatically detect attacks using machine learning [38], [64]. However, these methods also have significant detection error and suffer from generating too many false alerts [58], [34].

Threat Alert Triage. Ben-Asher *et al*. [27] did a study to investigate the effects of knowledge in detecting true attacks. They found that contextual knowledge about alert was more helpful in detection than cyber analysts experience and prior knowledge. Zhong *et al*. [68] mined past analysts' operation traces to generate a state machine for automated alert triage. Chyssler *et al*. [32] used a static filter with aggregation to reduce false alarm in IP network with the help of end-user involvement to adjust filter rules. There are many other proposed approaches to reduce the number of alerts such as careful configuration and improved classification methods [22], [45]; however, there are still too many threat alerts for the analysts to properly investigate [32]. To the best of our knowledge, NoDoze is the first system to leverage data provenance to automate the alert triage process in an enterprise without analysts involvement.

Provenance Analysis. A lot of work has been done to leverage data provenance for forensic analysis [26], [46], [56], [63], network debugging [23], [65], and access control [55]. Jiang *et al*. [39] used process coloring approach to identify the intrusion entry point and then use taint propagation approach to reduce log entries. Xie *et al*. [66] used high-level dependency information to detect malicious behaviour. However, this system only considered one event at a time without malicious behaviour propagation *i.e*. if ftp connects to some socket address which is not in their normal event database they will mark it as malicious. However, NoDoze considered the whole path i.e. the creation and ramification ftp-socket event for categorization by using anomaly score propagation algorithm. PrioTracker [49] accelerates the forward tracing by prioritizing abnormal events. Unlike PrioTracker, NoDoze focuses on triaging alerts and generating a more precise provenance graph. Furthermore, Priotracker only considers the abnormality of single events, it is not capable to distinguish similar events with different contexts in the dependency graph, such as $\mathcal{E}_1$ and $\mathcal{E}_2$ in our motivating example in Figure 2a.

Dependency graph compression techniques [47], [36], [30] are proposed to reduce the space overhead of provenance tracking. Provenance visualization [28] technique is also proposed to facilitate data provenance analysis. As such, these techniques do not remove any benign events for the efficient alert triage and investigation; however, NoDoze can use these techniques to further decrease the overhead of backward and forward tracing by removing redundant event.

XII. Conclusion

We develop NoDoze, a threat alert triage system that features historical and contextual information of generated alerts to automatically triage alerts. NoDoze uses a novel network diffusion algorithm to propagate anomaly scores in the dependency graphs of alerts and generates aggregate anomaly scores for each alert. These aggregate anomaly scores are then used by NoDoze to triage alerts. Our evaluation results show that our system substantially reduces the slog of investigating false alarms and accelerates the incident investigation with concise contextual alerting. NoDoze runtime overhead is low enough to be practical and can be deployed with any threat detection software.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful feedback. This work was supported in part by the National Science Foundation under contracts CNS-16-57534 and CNS-17-50024. This work was done while Wajih Ul Hassan and Shengjian Guo were interns under the supervision of Ding Li at NEC Labs America. Ding Li is the corresponding author of this paper. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of their employers or the sponsors.

References

[1] "Event Tracing," https://docs.microsoft.com/en-us/windows/desktop/ETW/event-tracing-portal.

[2] "Automated Incident Response: Respond to Every Alert," https://swimlane.com/blog/automated-incident-response-respond-every-alert/.

[3] "Endpoint Monitoring & Security," https://logrhythm.com/solutions/security/endpoint-threat-detection/.

[4] "How Many Alerts is Too Many to Handle?" https://www2.fireeye.com/StopTheNoise-IDC-Numbers-Game-Special-Report.html.

[5] "Insider Threat Detection," https://www.netwrix.com/insider_threat_detection.html.

[6] "New Research from Advanced Threat Analytics," https://prn.to/2uTiaK6.

[7] "Pass-the-hash attacks: Tools and Mitigation," https://www.sans.org/reading-room/whitepapers/testing/pass-the-hash-attacks-tools-mitigation-33283, 2010.

[8] "Automated Security Intelligence (ASI)," https://www.nec.com/en/global/techrep/journal/g16/n01/160110.html, 2018.

[9] "Dangerous and malicious file extensions," https://www.file-extensions.org/filetype/extension/name/dangerous-malicious-files, 2018.

[10] "Google Safe Browsing," https://developers.google.com/safe-browsing/v4/, 2018.

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

[12] "200,000+ Systems Affected by WannaCry Ransom Attack," https://www.statista.com/chart/9399/wannacry-cyber-attack-in-numbers/.

[13] "CVE-2014-6271," https://nvd.nist.gov/vuln/detail/CVE-2014-6271.

[14] "MS17-010 EternalBlue SMB Remote Windows Kernel Pool Corruption," https://www.rapid7.com/db/modules/exploit/windows/smb/ms17_010_eternalblue.

[15] "Target Missed Warnings in Epic Hack of Credit Card Data," https://bloom.bg/2KjElxM.

[16] "CVE-2008-0081," https://nvd.nist.gov/vuln/detail/CVE-2008-0081, 2018.

[17] "Persistent netcat backdoor," https://www.offensive-security.com/metasploit-unleashed/persistent-netcat-backdoor/, 2018.

[18] "Ransom.Wannacry," https://symc.ly/2NSK5Rg, 2018.

[19] "VirusTotal," https://www.virustotal.com/, 2018.

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

[21] S. Axelsson, "The base-rate fallacy and the difficulty of intrusion detection," *ACM Trans. Inf. Syst. Secur.*, 2000.

[22] D. Barbará and S. Jajodia, *Applications of data mining in computer security*. Springer Science & Business Media, 2002.

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

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

[25] A. Bates, B. Mood, M. Valafar, and K. Butler, "Towards secure provenance-based access control in cloud environments," in *CODASPY*, 2013.

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

[27] N. Ben-Asher and C. Gonzalez, "Effects of cyber security knowledge on attack detection," *Computers in Human Behavior*, vol. 48, 2015.

[28] S. P. Callahan, J. Freire, E. Santos, C. E. Scheidegger, C. T. Silva, and H. T. Vo, "Vistrails: Visualization meets data management," in *SIGMOD*. ACM, 2006.

[29] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Comput. Surv.*, 2009.

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

[31] T. Chen, L.-A. Tang, Y. Sun, Z. Chen, and K. Zhang, "Entity embedding-based anomaly detection for heterogeneous categorical events," in *IJCAI*, 2016.

[32] T. Chyssler, S. Burschka, M. Semling, T. Lingvall, and K. Burbeck, "Alarm reduction and correlation in intrusion detection systems." in *DIMVA*, 2004.

[33] J. Ellson, E. Gansner, L. Koutsofios, S. C. North, and G. Woodhull, "Graphvizopen source graph drawing tools," in *International Symposium on Graph Drawing*. Springer.

[34] R. Harang and A. Kott, "Burstiness of intrusion detection process: Empirical evidence and a modeling approach," *IEEE Transactions on Information Forensics and Security*, 2017.

[35] R. Hasan, R. Sion, and M. Winslett, "Preventing History Forgery with Secure Provenance," *Trans. Storage*.

[36] W. U. Hassan, M. Lemay, N. Aguse, A. Bates, and T. Moyer, "Towards Scalable Cluster Auditing through Grammatical Inference over Provenance Graphs," in *NDSS*, 2018.

[37] S. A. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion detection using sequences of system calls," *J. Comput. Secur.*, 1998.

[38] W. Hu, Y. Liao, and V. R. Vemuri, "Robust anomaly detection using support vector machines," in *ICML*, 2003.

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

[40] A. Kharraz, S. Arshad, C. Mulliner, W. K. Robertson, and E. Kirda, "Unveil: A large-scale, automated approach to detecting ransomware." in *USENIX Security Symposium*, 2016.

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

[42] M. Kloft and P. Laskov, "A poisoning attack against online anomaly detection," in *NIPS Workshop on Machine Learning in Adversarial Environments for Computer Security*, 2007.

[43] C. Kruegel, *Intrusion Detection and Correlation: Challenges and Solutions*. Springer-Verlag TELOS, 2004.

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

[45] A. Laszka, J. Lou, and Y. Vorobeychik, "Multi-defender strategic filtering against spear-phishing attacks." in *AAAI*, 2016.

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

[47] ——, "LogGC: garbage collecting audit log," in *CCS*, 2013.

[48] W. Lee and S. J. Stolfo, "Data mining approaches for intrusion detection," in *USENIX Security Symposium*, 1998.

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

[50] S. Ma, J. Zhai, F. Wang, K. H. Lee, X. Zhang, and D. Xu, "MPI: Multiple perspective attack investigation with semantic aware execution partitioning," in *USENIX Security Symposium*, 2017.

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

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

[53] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *ACM SIGMOD*, 2010.

[54] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer, "Provenance-aware storage systems," in *ATC*, 2006.

[55] J. Park, D. Nguyen, and R. Sandhu, "A provenance-based access control model," in *IEEE PST*, 2012.

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

[57] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, "A fast automaton-based method for detecting anomalous program behaviors," in *IEEE S&P*, 2001.

[58] R. Sommer and V. Paxson, "Outside the closed world: On using machine learning for network intrusion detection," in *IEEE Symposium on Security and Privacy*, 2010.

[59] G. P. Spathoulas and S. K. Katsikas, "Using a fuzzy inference system to reduce false positives in intrusion detection," in *International Conference on Systems, Signals and Image Processing*, 2009.

[60] S. J. Stolfo, S. Hershkop, L. H. Bui, R. Ferster, and K. Wang, "Anomaly detection in computer security and an application to file system accesses," in *International Symposium on Methodologies for Intelligent Systems*. Springer, 2005.

[61] G. Tandon and P. K. Chan, "On the learning of system call attributes for host-based anomaly detection," *International Journal on Artificial Intelligence Tools*, 2006.

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

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

[64] J. Wu, D. Peng, Z. Li, L. Zhao, and H. Ling, "Network intrusion detection based on a general regression neural network optimized by an improved artificial immune algorithm," *PloS one*, 2015.

[65] Y. Wu, A. Chen, A. Haeberlen, W. Zhou, and B. T. Loo, "Automated network repair with meta provenance," in *NSDI*, 2017.

[66] Y. Xie, D. Feng, Z. Tan, and J. Zhou, "Unifying intrusion detection and forensic analysis via provenance awareness," *Future Gener. Comput. Syst.*, 2016.

[67] 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.

[68] C. Zhong, J. Yen, P. Liu, and R. F. Erbacher, "Automate cybersecurity data triage by leveraging human analysts' cognitive process," in *IEEE BigDataSecurity*, 2016.