Analysis and Critique of Threat Detection and Alert Correlation in Intrusion Detection Systems

PhD Qualifier Exam: Written, Critique and Presentation

Sutanu Kumar Ghosh

Department of Computer Science
University of Illinois at Chicago

WCP Committee:
Prof. JON A. SOLWORTH
Prof. V.N.VENKATAKRISHNAN
Prof. OURI E. WOLFSON

February 2020

Abstract

Large enterprise networks are getting targeted by Advanced and Persistent Threats (APTs) increasingly in recent times. To detect these attacks, enterprises deploy a wide variety of Intrusion Detection Systems (IDS) or Security Information Event Management (SIEM) as a layer of protection against malicious attack groups. This report explains in detail two main aspects of Intrusion Detection Systems (IDS) or Threat Detection Softwares (TDS): **threat detection** and **alert correlation** and reviews four research pieces that utilizes audit records along with several intuitive techniques in order to detect threats and correlate alerts. The first two papers [1] [2] address the threat detection approaches and the last two papers [3] [4] focus on alert correlation techniques primarily. The aim of this report is to produce an in-depth study of these papers and figure out the advantages and the shortcomings of each of them. In the next section, I provide a detailed understanding of how an APT unfolds with a brief description of the four papers reviewed in this report. This is followed by the discussion of the four papers in detail including evaluations and limitations of each system. In the final section, I provide some general discussion on these systems along with a few future research insights.

I. Introduction

Nowadays Advanced and Persistent Threats (APTs) are increasingly plaguing large enterprise networks. Advanced and Persistent Threats (APTs) are very advanced and sophisticated cyber-attacks campaigns that consist of multiple stages and spans across a time period which is typically a few weeks to several months. What makes APTs such a vicious attack is that these APT campaigns are inflicted upon enterprise networks by means as simple as spearphishing emails. Once the APT actors successfully penetrates the network they uses a variety of advanced tools and stealthy softwares which enables them to persist in the network for weeks (or months) without getting detected. The goal of a successful APT campaign is to penetrate a network, persist on it, gather the necessary data and confidential information and exfiltrate it to some C2 server.

In a very detailed report (named APT1) [5] as published by Mandiant, a security firm discussed in detail the goals, activities, and effects of a global APT actor. These include stealing sensitive and confidential information worth hundreds of terabytes from at-least 141 organizations. The largest data theft from a single organization mentioned in that report was 6.5 terabytes of data from an APT campaign spanning over 10 months. Over the years, there has been a drastic increase in the number of APT attacks which involves some very powerful APT actors, including some nation-state actors. To understand the motivation and operations of the APT actors the Mandiant report also provided an APT life-cycle model [Figure 1] which helps one significantly to understand the multiple stages or attack steps involved in an APT campaign and collectively how they achieve their actors' goals and motives.

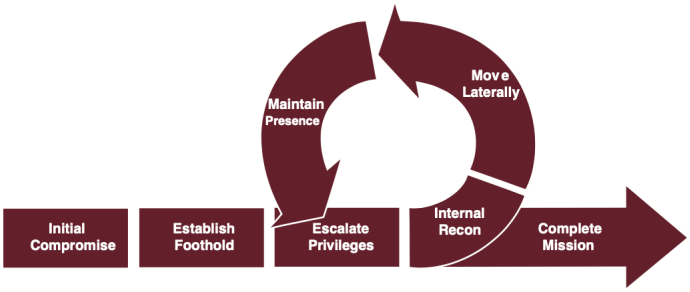

Fig. 1: APT Life Cycle

Since then, the sophistication and the stealthy attack techniques used in APTs are in rapid escalation. The most recent examples of a typical APT campaign include the Equifax data breach [6] which compromised the personal details of almost 150 million American citizens and the infamous DNC hack [7]. However even though APTs have evolved into more sophisticated attacks, the high-level steps of these attacks still conform to the life cycle in Figure 1. The main aim of the Intrusion Detection Systems (IDSs) is to detect these malicious attacks, either in real-time or in a forensic setting so that the security analysts can take preventive or mitigating measures. To that extent, enterprises deploy host-based or network-based IDS with each having its own set of limitations. In the first two papers: HERCULE [1] and SLEUTH [2] I have elaborated on the threat detection aspects of IDS and alert-correlation mechanisms are discussed by the means of PRIOTRACKER [3] and NODOZE [4]. Also, it is to be pointed out that SLEUTH [2] and HERCULE [1] acts as a complete threat detection system compared to the other two systems because these can detect threats as well as correlate alerts into meaningful attack scenarios simultaneously unlike the other two systems. The four systems reviewed in this report are primarily host-based IDS or host-based alert-correlation systems and all of them have a common input in the basic stages: *kernel audit logs*. The kernel acts as a bridge between system resources and software applications and has the ability to intercept every system call in an operating system and record them. All the four systems ingest these audit logs and perform some modifications and simplifications before using those in their respective architecture.

II. HERCULE: Attack Story Reconstruction via Community Discovery on Correlated Log Graph

HERCULE is a log-based intrusion detection system that is inspired by relationships in social networks. This multi-stage intrusion analysis system is modeled as a *community discovery problem*. HERCULE generates multi-dimensional weighted graphs by correlating audit logs, and detects "attack communities" within those weighted multi-dimensional graphs. The key idea is based on the observation that those log entries (the events contained in an audit log file) which are related to the attack stages have *dense and heavily-weighted* connections among themselves compared to the *sparse and lightly-weighted* connections among the benign log entries. The weighted graphs generated by HERCULE from the audit logs are analogous to social networks where people with similar interests, mutual friends or other similar features have closer and strong connections to each other.

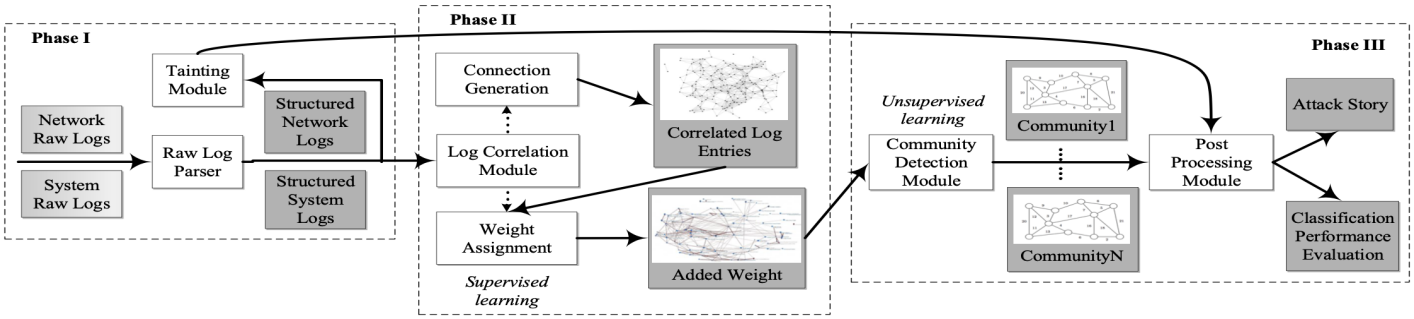

Fig. 2: HERCULE architecture

A. *Design and System Overview*

The architecture [Figure 2] of HERCULE can be differentiated into 3 different phases or modules each of which performs a series of non-trivial algorithms. The primary input to HERCULE is multiple raw logs both from the system (for e.g., process creation, deletion, file modifications, process executions, and others) and network activities (for e.g., DNS, HTTP, and others). All these raw logs [Table I] are fed into the first phase which is the **Raw Log Parser**. The parser parses each log and extracts a set of pre-defined features (which is referred to as a **data entity**) [Table II] that captures representative information of each log entry. A total of 20 different unique features are extracted by the Parser from each log entry at this stage. Each of these data entities are provided as an input to the Tainting Module and further into the second phase which is the **Log Correlation Module**. The **Tainting Module** parses the data entities and looks for any suspicious executable binary which is not in a whitelist by utilizing popular malware/virus platforms like Virustotal.

#	Logs	Provider
L1	DNS	Tshark
L2	WFP Connect	Auditd
L3	HTTP	Firefox
L4	Process create	Auditd
L5	Object access	Auditd
L6	Authentication	Syslogd

TABLE I: Logs Used

This module also looks for known malicious website accesses based on URL blacklists to identify and detect attack related logs. These initial attack related logs entries identified by the Tainting Module will be subsequently processed by the **Post Processing Module** in the last phase of HERCULE. The second phase or the Log Correlation Module is divided into 2 sub-modules: **Connection Generation** and **Weight Assignment**.

The **Connection Generation** sub-module takes the parsed data entities as input and produces undirected, unweighted, multidimensional graphs generated on the basis of inter/intra-log correlation. Just like in a social network, each log entry is treated as an individual or a node and each edge dimension as a type of relationship between any two individuals (or log entries). Accordingly, in an *n*-dimensional network, *G = (V, E, D)* where V is the set of nodes, *E* is a set of edges, and *D* is a set of dimensions- G forms $|V| \times |V| \times |D|$ a 3-dimensional boolean matrix M. Consequently, $M_{i,j,k} = 1$ indicates that between the log entry/nodes *i* and *j* there exists a correlation dimension *k*, otherwise $M_{i,j,k} = 0$. For each pair of log entries having one or more than one type of relationship between them, a multi-dimensional edge *e* connects those log entries (or nodes). The dimensions of this edge is defined as a uniform 29-feature vector $\vec{v} = [d_1, d_2, ...d_{29}]^T$ where a relationship between any two nodes (let's assume node *u* and node *v*) can be represented by the binary value of each dimension d_k. If

Field	Logs	Description
timestamp	L1-L6	event timestamp
q_domain	L1	DNS queried domain name
r_ip	L1	DNS resolved ip address
pid	L2, L4, L5	base-16 process id
ppid	L4	base-16 parent process id
pname	L2, L4, L5, L6	process name
h_ip	L2	host IP address
h_port	L2	host port number
d_ip	L2	destination IP address
d_port	L2	destination IP port
type	L3	request/response
get_q	L3	absolute path of GET
post_q	L3	absolute path of POST
res_code	L3	response code
h_domain	L3	host domain name
referer	L3	referer of requested URI
res_loc	L3	location to redirect
acct	L5	principle of this access
objname	L5	object name
info	L6	Authentication information

TABLE II: Features extracted by Parser

any two nodes or log entries can be correlated by *k*-th relationship then $d_k = 1$, otherwise $d_k = 0$. The 29-feature vectors are summarised below in Table III. The key intuition to define these 29 feature vectors is to capture any causal relationship between any two nodes or log entries. For two nodes/log entries, *u* and *v* the following set of feature vectors are defined: d_1 is used to model the time difference between nodes *u* and *v*. This threshold value t can be customized to determine whether two nodes are temporally correlated or not. d_2 and d_{13} checks if nodes *u* and *v* shares the same process id or process name to infer any implicit/explicit correlation among them. d_3 and d_4 verifiy whether *u* and *v* share the same destination IP and Port as they should have a high degree of correlation if they communicate to the same IP addresses. d_5, d_6, d_7, and d_8 capture the causality relations of different web page visits within HTTP logs. d_9, d_{10}, and d_{11} checks if nodes *u* and *v* share the same parent process id. Similarly, the other features also compare a variety of different parameters like object names accessed by *u* and *v*, inbound and outbound network traffic from DNS queries and several other parameters. After the feature vectors are determined for each edge, connections among nodes are generated based on these relationship values from the feature vectors. The inputs to the connection generation algorithm are the feature vectors from Table III and all the parsed data entities. This algorithm iterates over all possible feature values between log pairs and generates edges with at least one feature vector value being non-zero.

D	Feature
d_1	$\delta(u.timestamp, v.timestamp) < t$
d_2	$u.pid = v.pid$
d_3	$u.d\_ip = v.d\_ip$
d_4	$u.d\_port = v.d\_port$
d_5	$u.referer = v.referer$
d_6	$u.host = v.host$
d_7	$u.referer = v.host$
d_8	$u.host = v.referer$
d_9	$u.ppid = v.ppid$
d_{10}	$u.ppid = v.pid$
d_{11}	$u.pid = v.ppid$
d_{12}	$u.objname = v.objname$
d_{13}	$u.pname = v.pname$
d_{14}	$u.r\_ip = v.d\_ip$
d_{15}	$u.d\_ip = v.r\_ip$
d_{16}	$u.q\_domain = v.h\_domain$
d_{17}	$u.h\_domain = v.q\_domain$
d_{18}	$u.q\_domain = v.referer$
d_{19}	$u.referer = v.q\_domain$
d_{20}	$u.q\_domain = v.res\_loc$
d_{21}	$u.res\_loc = v.q\_domain$
d_{22}	$u.getq = v.pname$
d_{23}	$u.pname = v.getq$
d_{24}	$u.get\_q = v.objname$
d_{25}	$u.objname = v.get\_q$
d_{26}	$u.pname = v.objname$
d_{27}	$u.objname = v.pname$
d_{28}	$u.r\_ip = v.h\_ip$
d_{29}	$u.h\_ip = v.r\_ip$

TABLE III: Feature Vectors

In the **Weight Assignment** sub-module, a weight assignment algorithm assigns different weights to edges that have different edge feature values. The key intuition behind this algorithm is the disadvantages of applying Community Detection algorithm on unweighted multi-dimensional graph generated from the Connection Generation sub-module. By formal means, within two community clusters of nodes *A* (attack related log entries) and *B* (benign log entries) in a multi-dimensional unweighted graph *G*, it is expected to get $|e_A| \gg |e_{AB}|$ and $|e_B| \gg |e_{AB}|$ where $|e_A|$, $|e_B|$, and $|e_{AB}|$ denotes the number of edges in the clusters A and B and in between A and B respectively. In general, there exists several cases of log entries which are attack related but also have a significant connections with benign entries. Cases like these reduces the efficacy of the community detection algorithms which mainly aims to maximize the intra-cluster density and minimize the inter-cluster density. Another key observation made is that the

feature values between $\vec{e_{AB}}$, $\vec{e_A}$, and $\vec{e_B}$ (which denotes the edge vectors of e_{AB}, e_A, and e_B respectively) are significantly different. Under this observation, the weight assignment algorithm is implemented to assign different weights to edges with different edge feature values so that the following inequality holds: $w_A \cdot |e_A| > w_{AB} \cdot |e_{AB}|$ (w_A is the weight assigned for edges in e_A and w_{AB} is the weight assigned for edges in e_{AB}). This algorithm tries to learn a global weight vector $\vec{a}$ that can be applied intuitively on each edge. A sigmoid function S is used to map the dot product to bounded real number range [0,1] as the finalized weight assignment value on each edge is:

$$w = S(\sum_{i=1}^{k} a_i \cdot e_i) = \frac{1}{1 + e^{-\sum_{i=1}^{k} a_i \cdot e_i}}$$

Then the graph is multi-dimensional unweighted graph is transformed into a weighted graph *WG*. To determine the optimal value of w the *training phase* and the *testing phase* is defined as follows: Given n unweighted graphs $G_1, G_2, ...G_n$ for each $l (l \in [1, n])$ the *training phase* of the weight assignment looks for a best assignment weight vector $\vec{a_k}$ for $G_1, ..., G_{l-1}, G_{l+1}, ..., G_n$ and the *testing phase* takes the dot product of weight vector $\vec{a_l}$ and all edge vectors $\vec{e}$ to generate a weighted graph WG_l. Several algorithms were used and compared for the optimal weight assignment value. As a simple and straight forward approach, Feature Weight Summation is calculated in which the algorithm considers each feature value of an edge with the same "importance". However, there were performance issues by implementing this approach. To overcome the limitations two supervised learning techniques were implemented namely, **Logistic Regression** and **Support Vector Machines(SVM)**. These two classification-based learning methods classified edges into two classes(e_A, e_B vs e_{AB}) on the basis of decision boundary. But, the weight vectors produced by these two methods were not globally optimum as a result of which the value assigned to edges of e_A and e_B were not maximized and the weight values assigned to e_{AB} were not guaranteed to be minimized. Therefore a new approach was designed and implemented which transformed the weight assignment into a **quadratic optimization** problem. A trade-off parameter is introduced to balance between the attack-related nodes and benign nodes. A regularizer is also implemented to overcome the Overfitting problem of the machine learning techniques which ensures that the output weight vector $\vec{a}$ is theoretically global optimum. Since the optimization is constrained, the previously used sigmoid function is not leveraged again to map the dot product to [0,1].

The third and the final phase is the **Community Detection** phase in which in which the correlated multi-dimensional weighted graph is taken as input and communities are generated by using the Louvain algorithm [8]. The initialization of communities is done from the weighted graphs by applying Louvain algorithm in two different phases to finally build a new network. Then, the detected communities are fed into the **Post Processing Module**. This module also takes input the log entries tainted in phase 1 (by the tainting module) and classifies the communities that contains tainted nodes/entries as malicious and rest as benign. Finally, the reconstructed attack phases are produced as output.

B. Evaluation

HERCULE is evaluated against 15 real-world APT attacks which were recreated with some minor modification as necessary since it is very hard to exactly replicate the APT attacks. To determine its scalability a two-week long experiment was also conducted which mostly contained normal benign user activities such as browsing websites, downloading and updating softwares, watching videos along with three APT attacks. On an average, the accuracy with which HERCULE classified the log entries within any identified community as malicious or benign is around 89.87% with a low false positive rate. A detailed analysis of the results of all the 16 APT attack scenarios is provided in Table IV. In Figure 3 the communities marked in red are the identified group of malicious log entries and the other communities are marked with different random colors.

APT Keyword	Initial Tactics	CVE	Post Exploitation	Target	Acc	FP
Black Vine 1	Watering hole	2012-4792	Keylogger	Win	0.846	0.0012
Black Vine 2	Email attachment	2014-0322	Exfiltrate files	Win	0.834	0.0023
Attack on Aerospace	Watering hole	2015-5122	Network sweeping	Win	0.810	0.0018
Tibetan and HK	Email google drive links	2014-4114	Exfiltrate files	Win	0.886	0.0013
Op-DeputyDog	iframe background running	2013-3893	Escalate privilege	Win	0.877	0.0024
Russian Campaign	Controlled Website	2015-3043	Download backdoor	Win	0.833	0.0023
Op-Clandestine Fox	Email compromised website	2014-1776	Rename payload	Win	0.857	0.0026
Cylance SPEAR Team	Email attachment	2012-0158	Browsing files	Win	0.826	0.0016
APT on Taiwan	Email attachment	N/A	Rename payload	Win	0.819	0.0010
Op-Tropic Trooper	Email attachment	2010-3333	Download tools	Win	0.812	0.0006
Op-Tropic Trooper	Email attachment	2012-0158	Keylogger	Win	0.863	0.0090
Hacking Team	Email with file link	2015-5119	Download backdoor	Win	0.859	0.0058
Russian Campaign	Email attachment	2008-5499	Download backdoor	Linux	0.850	0.0017
Op-DeputyDog	Email compromised website	N/A	Brute force login	Linux	0.899	0.0060
SeaDuke	Email trojaned-ware	N/A	Add bad user	Linux	0.874	0.0012
Two weeks	Combined APTs	N/A	N/A	Win	0.736	0.0126

TABLE IV: Evaluation Results

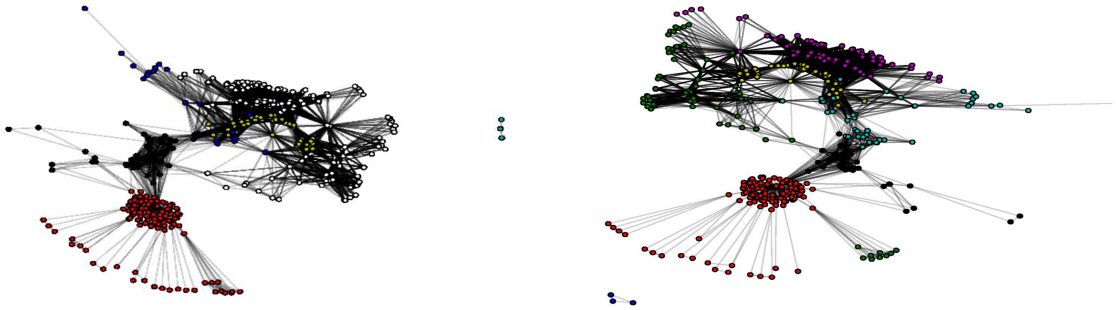

Fig. 3: Community Distribution (Red nodes denote malicious community)

C. *Limitations*

In this section, I will discuss the contributions and critiques of HERCULE from my point of view. The main contribution of this paper is developing and implementing a novel automated multi-stage intrusion analysis system which facilitates attack scenario reconstruction from multiple correlated logs. It leverages community discovery techniques to correlate attack steps and detect an attack. This system is then evaluated against a wide range of different real-world APT attacks. Although HERCULE performs exceptionally well against these manually generated APT attack scenarios, there are a few pitfalls which can be discussed in the light of this Written Critique. Starting with the technical shortcomings, it is important to note here that even though HERCULE builds up weighted multi-dimensional graphs to detect attack related log entries, a significant amount of the detection depends upon the Tainting module which is basically a manually updated whitelist. This paper explicitly states that along with VirusTotal [9], it leverages BitBlaze [10] to analyze any suspicious executable binary that appeared in the log entries. Now there are two limitations to this: first, in the majority of the APT attacks, attackers tend to clean up their attack traces after the completion of the attack. Hence, the malicious executable binaries used by the attackers would be removed (deleted) from the system even before HERCULE could feed it into Tainting Module. The name of a suspicious executable appearing in the log entries does not guarantee that the same executable would still be present in the system to analyze when HERCULE is running. And secondly, in recent years, there have been a wide range of dynamic malware analysis systems like [11][12][13][14], each of which has its own set of pros and cons compared to the others. A comparative study between such popular dynamic malware analysis systems would be more efficient instead of fielding any random system. Moreover, it is evident that these attacks are evolving very rapidly and by leveraging online malware analysis platforms like VirusTotal, there remains a very high chance that the attacks would not be detected in due time.

Finally, the output produced by HERCULE as can be seen in Figure 3 still requires a significant manual analysis to reconstruct the attack story from the detected malicious communities. It is not feasible to infer meaningful steps about the attacks from the community detection graphs as the graphs are too congested. This particular limitation has been resolved in other systems [2][4] which are discussed later in this report. Those systems produce concise and more meaningful attack graphs than compared to HERCULE. The key intuition to generate concise, meaningful graphs is that the cyber analysts should not spend much time inferring the attack steps while analyzing these graphs. This limitation is evident in HERCULE from the fact that the paper could not even include the graph for the two-week-long custom attack scenario experiment as it is too dense to be shown in the paper. This shows that if this system is implemented in real-world enterprise networks (which consists of billions of activities stretched over a far more elongated period of time) it would not perform as efficiently as compared to the manually crafted attack scenarios.

III. SLEUTH: Real-time Attack Scenario Reconstruction from COTS Audit Data

SLEUTH is an advanced intrusion detection system [2] which enables the real-time reconstruction of attack-scenarios on an enterprise network. The billions of audit logs in an enterprise network are scaled to detect the real-time detection of attacks in a platform-neutral, main-memory based, dependency graph abstraction model based on the audit logs. There are a number of key challenges of real-time attack detection which have been solved by a series of novel approaches. SLEUTH addresses the key problem of an efficient event storage and event analysis (from the audit logs) based on the main memory by the development of a compact main-memory dependence graph representation, which performs far better than popular graph databases such as Neo4J [15] or Titan [16]. SLEUTH implements a *tag-based approach* to detect and identify entities and events that are likely related to attack steps. Tags helps to prioritize and focus the analysis by incorporating an assessment of *trustworthiness* and *sensitivity* of the data and the processes. This assessment is based on *Provenance, Prior system knowledge, Behaviour* derived from the audit logs. Further SLEUTH leverages a customizable policy framework that can raise detection alerts based on these tags. Finally, a *backward analysis* and *forward analysis* reveals the full scale of the impact of the adversarial actions in a compact graph.

A. *Design and System Overview*

The architecture of SLEUTH[1] [Figure 4] can be divided into four main parts: First, the main-memory based dependence graph generation from the audit logs. Second, the Tag and Policy-based attack detection based upon a set of customizable policies. And finally, Root cause and Impact analysis followed by the reconstruction of the attack scenario in the form of scenario graphs. To facilitate real-time analysis, the dependencies are stored in a graph data structure. One other reason to use main-memory dependence graph representation is that many graph algorithm applications are limited unless the main memory is large enough to process the data. In an enterprise network, the number of events can easily range in billions to tens of billions per day which would require a humongous amount of main memory, which is infeasible. The graph databases optimized for main-memory performance like STINGER [17] and NetworkX [18] use about 250 bytes and 3KB per edge respectively. Compared to that, SLEUTH implements a more space-efficient compact dependence graph representation that uses only 10 bytes per edge. This dependency graph representation is a *per-host data structure* which is mainly optimized for intra-host reference. The graph mainly represents two types of entities: *subjects*, which are processes, and *objects*, which are entities such as files, pipes, and network connections. There are several attributes associated with Subject such as process ids (or pid), command line, owner, code tags and data tags. For Objects, the attributes include name, type, owner and tags. Events recorded in audit logs are mapped as labeled edges between subjects and objects or between two subjects. A detailed explanation of the algorithms used to reduce the storage space requirements for each event, subject or object is published in [19].

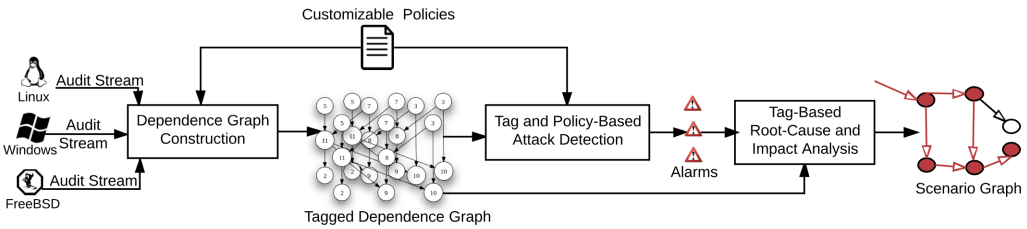

Fig. 4: SLEUTH System Overview

Tags play a crucial role in attack detection in SLEUTH. Each audited event recorded in the audit logs is interpreted in the context of these tags to determine whether it contributes to an attack or not. Moreover, tags helps to expedite the forward and backward analysis. Further, tags play a very important role in the attack scenario reconstruction phase by pruning out huge amounts of benign data that does not meaningfully contribute towards an attack. A brief description of the tag design is as follows. Following tags are defined as *trustworthiness tags (t-tags)*:

- *Benign authentic tags*: Assigned to data or code which are received from sources trusted to be benign and whose authenticity can be verified.
- *Benign tag*: Lower level of trust compared to *benign authentic*. The data/code is still supposedly benign.
- *Unknown tag*: Assigned to data/code whose information is relatively unknown.

Policies are used to define what sources are benign, which in simple terms can be used as a whitelist. If no policy is applicable to a source then its *t-tag* is set to *unknown* (More about policies later). Following are the tag definitions for *confidentiality tags (c-tags)*:

- *Secret*: Confidential and highly sensitive data, such as login credentials or private keys
- *Sensitive*: Data which does not facilitate a direct way to an attacker to gain access to a system, but its disclosure can lead to several security impacts.
- *Private*: Data which if leaked can lead to privacy concerns but not necessarily lead to any security impact.
- *Public*: Data which is publicly available.

Pre-existing objects and subjects are assigned some initial values based on the *tag initialization policies*. External objects such as remote network connections are also assigned initial tags according to policies. The other subjects and objects created when the system is running are assigned tags based on the *tag propagation policies*. A subject/process is given two types of integrity tags: *code trustworthiness tags (code t-tags)* and *data trustworthiness tags (data t-tags)* which facilitates attack detection.

A set of policies or rules are defined based on these tags to detect attacks. These policies are customizable as per the host system and are flexible enough to focus on any particular application or process at any given time. Some of the most important attack detection policies are defined below:

- *Untrusted Code Execution*: When a subject with a higher code t-tag loads or executes an object with a lower t-tag, this policy is triggered and subsequently an alarm is raised.
- *Modification by subject with lower code t-tags*: When a subject with a lower code t-tag modifies (change name, permission and others) an object with a higher t-tag then this policy alarm is raised.

[1]SLEUTH stands for (attack) Scenario LinkagE Using provenance Tracking of Host audit data

Event	Direction	Alarm Trigger	Tag Trigger
define			$init$
read	O→S	$read$	$propRd$
load, execve	O→S	$exec$	$propEx$
write	S→O	$write$	$propWr$
rm, rename	S→O	$write$	
chmod, chown	S→O	$write, modify$	
setuid	S→S		$propSu$

TABLE V: Edges with policy trigger points. In the direction column, S indicates subject, and O indicates object. The next two columns indicate trigger points for detection policies and tag setting policies.

- *Confidential data leak*: When untrusted subjects exfiltrate sensitive data this alarm is raised. A typical scenario would be network writes by subjects with a *Sensitive* c-tag and *Unknown* code t-tag.
- *Preparation of untrusted data for execution*: This alarm is raised when an operation by a subject with *Unknown* code t-tag, is followed in conjunction with some library loading operations.

The APT attacks in general do not involve the execution of the untrusted code in the first step itself. The attacker would download and execute the malicious code by some means, or change some memory permissions containing the malicious code and eventually execute the untrusted code. The policies are designed in such a way that eventually one of the policies could detect the attack.

For the tag assignment, tag propagation and attack detection purposes a customizable and flexible policy framework is also implemented. The policies can be defined by the means of simple rule-based notation encoded as C++ functions. An example of such rule-based notation which is used to trigger an alert for Untrusted Execution is as follows:

$$exec(s, o) : o.ttag < benign \rightarrow alert(\text{"UntrustedExec"})$$

This particular rule is triggered when a subject s executes a file/object o with a t-tag lower than $benign$ and alert named UntrustedExec is raised. These rules are generally incorporated with events, and include conditions on the subject and object attributes involved within the event. A variety of attributes can be included in these conditions, such as Perl syntax regular expressions are used to match object names and subject command lines. The t-tags and c-tags values of subjects and objects can be compared with certain thresholds. Moreover, the ownership of subjects and objects or the permissions associated with an object or an event can be included in these attributes. Different policies have different effects. The main function of a detection policy is to raise an alarm. The tag initialization or tag propagation policies update the tags associated with subject or object in an event. In order to induce a finer degree of control over the order in which the policies are checked, a *trigger point* is defined instead of events. The trigger points defined in the policy framework is shown in Table V. It provides a level of indirection that enables sharing of policies across a set of similar events. When any particular event is encountered, all detection policies associated with its alarm trigger are executed and checked only when the destination subject or object tag is about to change. After this, policies associated with the event's tag triggers are checked in the order in which they are specified. As soon as a matching rule is found, the tags defined by that rule are assigned to the destination subject or object in that event and the remaining policies are not checked.

The **Tag Initialization Policies** are invoked at the *init* trigger to initialize tags for new objects, or pre-existing objects. In a default setting, when a subject creates a new object, this object inherits the subject's tags. This can be overridden using the Tag Initialization policies. A simple example of such a policy is as follows:

$$init(o) : o.type == \text{FILE} \rightarrow o.ttag = \text{BENIGN\_AUTH}, o.ctag = \text{PUBLIC}$$

This rule specifies that if the object type is a File, then t-tag and the c-tag of that object are initialized to BENIGN_AUTH and PUBLIC respectively. Several such initialization policies can be defined to initialize different objects or subjects.

The **Tag Propagation Policies** are used to override the default tag propagation values. Similar to tag initialization polices, different tag propagation policies can be defined for a different group of related events as indicated in Table V. Following is an example of such a policy which prevents over-tainting from .bash_history file which is repeatedly read and written by an application each time it is invoked.

$$propRd(s, o) : match(o.name, \text{"\textbackslash.bash\_history\$"}) \rightarrow skip$$

In the final phase of SLEUTH, a backward and forward analysis is performed from a particular node (from which an alarm is raised) to determine the root cause and impact analysis of an attack. The primary aim of **Backward Analysis** is to zoom in and detect the entry point of an attack among the hundreds of millions of nodes in the dependency graph. A naive backward search can lead to performance issues and determining multiple entry nodes leading to false positives. The key intuition used here is that Tags are used to overcome both the challenges. A tag value of *unknown* on a node increases the likelihood of that node being a part of an attack campaign rather than the neighbouring nodes with *benign* tags. The backward search in SLEUTH is defined as an instance of shortest path problem leveraging the Dijkstra's algorithm. Tags are used to define edge

Dataset	Drop&Ld	Intel Gather	Backdoor Insert	Priv Escl	Data Exfil	Cleanup
W-1	✓	✓			✓	✓
W-2	✓	✓	✓		✓	✓
L-1	✓	✓	✓		✓	✓
L-2	✓	✓	✓	✓	✓	✓
L-3	✓	✓	✓	✓	✓	✓
W-1			✓		✓	
F-2	✓	✓	✓		✓	
F-3	✓	✓			✓	

TABLE VI: SLEUTH Evaluation Results

costs which in turn guide the backward search along relevant paths leaving out the benign paths. Information flows among benign nodes which are not part of the attack are assigned a very high-cost value. Edges having dependency from a node with *unknown* code or data t-tag to a node with *benign* code or data t-tag are assigned a cost value of 0 and likewise.

The second part of this Tag-based bi-directional analysis is the **Forward Analysis** which determines the impact of an attack campaign starting from the entry point (or node). A custom distance threshold value d_{th} is defined to prune out the nodes which are deemed too far from the suspect nodes. This value can be interactively adjusted by an analyst. The same cost metric as backward analysis is used for the forward analysis too. Finally, the output of forward analysis is subjected to a series of algorithms to transform the final **scenario graphs** into compact and simple graphs which can easily interpreted by the analysts. These algorithms performs a variety of non-trivial simplifications which include pruning of unsuspicious nodes, merging entities with the same names and filtering out repeated reads or writes between the same set of entities. These algorithms are further described in detail in [19]

B. Evaluation

The main components of SLEUTH: dependency graph generation, policy framework, attack detection mechanisms are all implemented in C++ consisting of 9.5KLoC approximately. Apart from this, the forward and backward analysis systems for scenario graph reconstruction and presentation are implemented in Python in about 1.6KLoC. SLEUTH is evaluated in an adversarial engagement setup where attack campaigns were carried out by Red teams as a part of the DARPA Transparent Computing (TC) program. Attack datasets were collected on two Windows, three Linux and three FreeBSD machines spanning over a period of 358 hours and contained 73 million events. Since this was an adversarial engagement, no prior knowledge of the red team attacks were provided beforehand. SLEUTH was able to detect the majority of the red team attacks [Table VI] which included drop and load activities, backdoor insertion, privilege escalation, data exfiltration by means of C&C servers and cleanup of attack traces from the system. SLEUTH successfully detected the key entry and exit points, key files accessed, and also missed out on a couple of entities. The scenario graphs generated during this campaign are also very precise and detailed reflecting the steps of the attacks as performed by the red teams [Figure 5]. The performance of this system in a benign environment is highly impressive. Audit data was collected from four Ubuntu Linux servers spanning over 3 to 5 days. The key focus was on software updates and system upgrades in this experiment resulting to changes in 110 packages along with thousands of binary and script files also being updated. With some changes to the policy framework, all the updates and downloads were marked as benign leading to no alarms or false positives being generated.

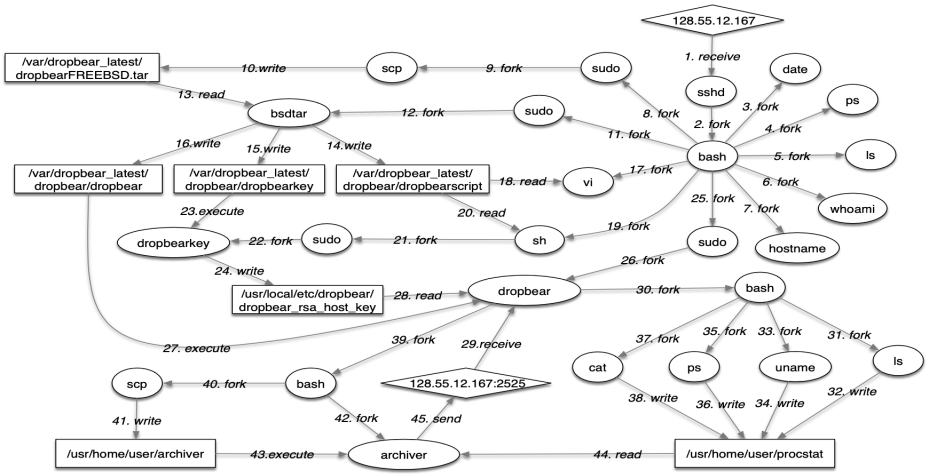

Fig. 5: Scenario graph reconstructed from one of the attack campaigns

C. *Limitations*

Before discussing a couple of drawbacks of this system, I will point out, in brief, the major contributions of SLEUTH. The key contribution of this paper is the design and implementation of an efficient real-time intrusion detection system that leverages main memory dependence graph data model. Further, efficient tag-based techniques are implemented for attack detection and reconstruction purposes. A customizable policy framework for tag propagation and tag initialization is also implemented for these tag-based techniques to work efficiently. Another distinctive feature of this system is that SLEUTH can be implemented by an analyst without *detailed application-specific knowledge* compared to other intrusion detection systems.

Focusing to the drawbacks of the system, I feel there are a few shortcomings which are pertinent to the application purposes of this system. As mentioned earlier, SLEUTH is a *per-host data structure*. The scalability of this system to implement across inter-host architecture is not feasible. As a result, it may lead to attacks being misreported if an attacker tries to achieve lateral movement across different hosts. To the best of my knowledge, there are no host-based distributed real-time intrusion detection systems commercially available. To this extent, deploying SLEUTH in enterprise networks would come with a trade-off. It is fairly simple to set-up and implement this system in an enterprise network which would consist of hundreds of hosts.

Another disadvantage of SLEUTH is the dependency explosion which is potentially caused in long-running attacks where normal activities get connected to malicious activities leading to graph explosion. For long-running attacks this system can produce graphs that may contain numerous benign nodes, thus reducing the efficacy of the system. This is mainly due to naive tag propagation and get can be mitigated with the help of a more robust tag propagation techniques. The problem of dependence explosion has been resolved later by Milajerdi *et al.* [20] in HOLMES which inherently uses the same causality tracking, provenance graph generation methodologies of SLEUTH. However, HOLMES introduces a notion of *minimum ancestral cover* to overcome dependence explosion. Moreover, NODOZE [4] (discussed later) tackles the dependence explosion problem by leveraging *behavioural execution partitioning*.

IV. Towards a Timely Causality Analysis for Enterprise Security

In this paper, an alert-correlation system named **PRIOTRACKER** [3] is developed to distinguish the normal benign events from the malicious ones by assigning a rareness score to each event and comparing it to a **reference model** which records the regular activities in an enterprise computer system. PRIOTRACKER leverages a backward and forward causality tracker to help investigate the malicious causal dependencies among the events. Further, this *attack causality analysis* has been modeled as an optimization problem to detect the maximum number of malicious events within a certain time period. A *priority score* for each event is also calculated based on its rareness score and certain other features (from the causality graph) and the event with the highest priority score is analyzed. Weights are also assigned to these features (more on this later) by means of machine learning algorithms to reveal the maximum number of rare events before a given deadline.

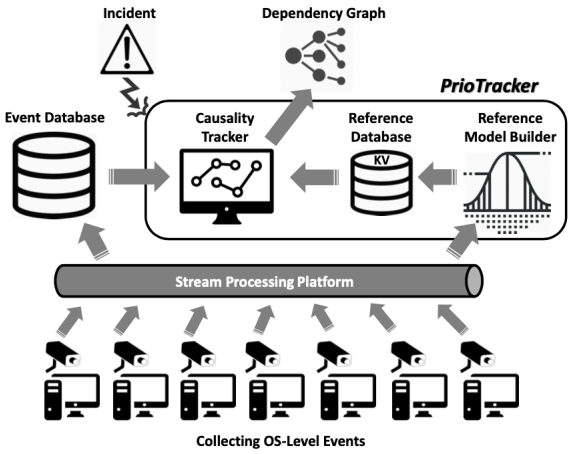

Fig. 6: PRIOTRACKER Architecture

A. *Design and System Overview*

The architecture of PRIOTRACKER [Figure 6] can be differentiated into three main parts: a priority-based event **causality tracker**, a **reference database** and a **reference model builder**. Three different types of low-level system events are collected from Windows and Linux machines using ETW and kernel audit respectively. These are file events (such as file read, write and execute), process events (process create and destroy) and network events (socket create, destroy, read, and write). The **Causality Tracker** starts to build a dependency graph between OS-level events from an intrusion alert raised by the existing detection system. This dependency graph generation process is very similar to BackTracker [21] but modified to prioritize abnormal

or malicious event tracking. The key intuition here is that the causality tracker maintains a priority queue data structure that contains all the events from their respective alerts to be analyzed. After an alarm is raised, the priority score of that event is calculated and is further added to the priority queue. PRIOTRACKER then iteratively analyzes each event based on its priority score until the queue is empty or a time threshold (T_{limit}) is reached. As it retrieves an event from the head of the queue and adds into a result graph, an algorithm calculates its causal dependencies, returns a set of events (which that particular event has dependencies with), calculate the priority scores of those events and adds them back into the priority queue. The value of the priority score depends on a few different important factors. First, a **Rareness Score** is calculated which is based on the frequency of the occurrence of events across different hosts from the **Reference Model** and the **Reference Database**. The rareness score is defined as follows:

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

The $ref(e)$ is the reference score of the event e which will be discussed later on. Next, a **Fanout Score** is implemented on the basis of pruning out read-only files and taking into account the write-only files. This heuristic is also motivated from the BackTracker [21]. The fanout score is defined as follows:

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

The **Priority Score** of an event can now be defined as the weighted sum of the rareness score and the fanout score of that event e.

$$Priority(e) = \alpha \times rs(e) + \beta \times fs(e) \qquad (1)$$

The value of α and β is an optimization problem to maximize the result of the objection function for a given set of events.

$$\max\ f(E, (\alpha, \beta) = \sum_{e \in E} EdgeCount_{\theta}(PrioTrack_{(\alpha,\beta)}(e, T_{limit})) \qquad (2)$$

s.t. $0 \leq \alpha \leq 1, \alpha + \beta = 1$

The α and β here are the weight values for rareness and fanout scores respectively. $PrioTrack$ is the dependency algorithm defined earlier and $EdgeCount$ is a function which count counts the number of edges in the graph whose rareness score is greater than a customizable threshold θ.

Next, the **Reference Model** is implemented which collects data from 54 Linux and 96 Windows machines used daily in an enterprise system to quantify the rareness of system events and help distinguish the anomalies from benign system events. The data is collected and modeled in the following efficient ways:

- The data collected from the 150 machines are subjected to *Mixed Membership Community and Role Model* (MMCR) [22] to segregate machines from different departments in the company into three different communities.
- The file, process and network events collected from these machines are abstracted using Backus-Naur form (BNF) [Figure 7]
- A notion of a *time window* is implemented which maintains a counter that is increased upon the repeated occurrence of an event on the same host within that time period. These repeated occurrences are considered as once to withstand attacks that include repeated malicious activities with a burst of events which tricks a benign system to consider them as common behaviours. The time window here is configured to be one week worth of time as enterprises are generally operated on a weekly basis.

⟨*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. 7: Event Abstraction using BNF

With the help of these features, the **Reference Score** of an event e is defined as its accumulative occurrence on all homogeneous hosts for all weeks.

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

where $hosts$ are the set of similar machines (detected by MMCR) and $weeks$ is the set of weeks of data collected. Further $count$ is defined as

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

Next, Hill Climbing algorithm is leveraged to optimize Equation 3. This algorithm gradually improves the quality of weight selection by implementing a feedback loop. The algorithm takes a set of starting events E and initial weight vectors (α and β) as inputs. For the starting event set E, 1113 randomly selected system events within the last 10 months were selected. After every iteration, the algorithm adjusts an individual element in weight vectors and compares whether this change improves the value of the objective function $f(E, (\alpha, \beta)$. A positive change is accepted and this process continues until no positive change could be found. Here, the algorithm optimized the weight vectors to a value of $\alpha = 0.27$ and $\beta = 0.73$

B. *Evaluation*

The priority-based dependency tracker in PRIOTRACKER have been developed in Java consisting of 20KLoC and the Reference Model is also implemented in Java in about 10KLoC. The Causality tracker frequently accesses the reference database in order to determine the reference score of associated events. As a result, RocksDB which enables in-memory key-value store for fast access and data persistence is leveraged here. PRIOTRACKER is evaluated on a dataset which is 1TB in size and consists of 2.5 billion events approximately collected from the 150 hosts over one week. Moreover, eight different attacks are crafted associated with *noise* or normal system operations to determine the efficacy of PRIOTRACKER. The results of PRIOTRACKER are compared with a baseline forward tracker from BackTracker [21]. The results are summarized in Table VII. It is evident that PRIOTRACKER performs much better than compared to the baseline tracker and is also able to successfully detect all the attacks without missing any one of them. The difference in detection time between these two systems is also significant. PRIOTRACKER is able to detect attack traces in a very short time which is almost 2 magnitudes smaller than the same in some cases. A reduced version of Forward Tracking Graph also shows the attack steps in Figure 8.

Attack Case	Baseline			PrioTracker		
	Runtime	FNR	Critical Events	Runtime	FNR	Critical Events
Data Theft	6.29s	0%	13	1.55s	0%	13
Phishing Email	45.90s	0%	148	27.51s	0%	148
Shellshock	37.45s	0%	25	9.13s	0%	25
Netcat Backdoor	6.85s	0%	14	0.88s	0%	14
Cheating Student	NA	62%	14	24m47s	0%	37
Illegal Storage	15m31s	0%	12	12m39s	0%	12
wget-gcc	18m31s	0%	25	5m37s	0%	25
password-gzip-scp	NA	40%	9	57s	0%	15

TABLE VII: Evaluation Results of PRIOTRACKER. FNR = False Negative Ratio

C. *Limitations*

In brief, the key contributions of PRIOTRACKER include design and implementation of a novel priority-based causality tracker that distinguishes malicious events from the benign ones by calculating a priority score based upon the rareness and the fanout of that event. Hill Climbing algorithm is applied to these features to optimize the value of the priority score. Moreover, a customizable Reference Model is also implemented to observe and model the OS-level events from hosts in an enterprise to facilitate the computation of priority score of events. Also, the dataset used for evaluation purposes is significantly large and contains audit data from 150 different (Windows and Linux) machines with a wide variety of attacks being performed in it. There are however a few drawbacks to this system which can potentially jeopardize the efficacy of PRIOTRACKER. First,

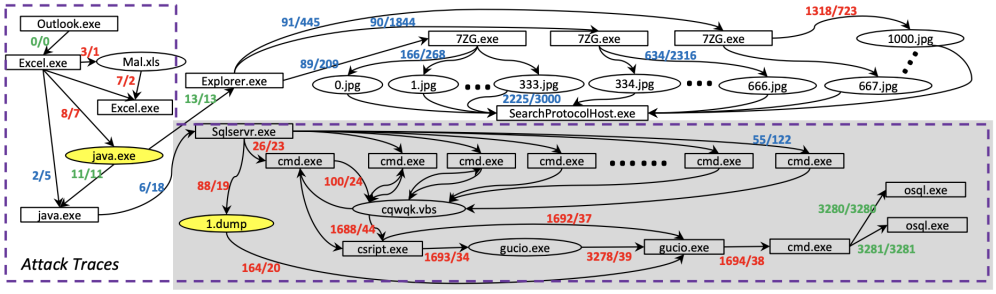

Fig. 8: Reduced Version of Forward Tracking Graph for Email Phishing Attack

this whole architecture starts from a detection signal or an intrusion alert raised by an existing intrusion detection system. It proceeds to query the reference database and compute that particular event's priority score and generate the causality graph. With the advent of all these zero-day attacks, it is highly possible that the existing intrusion detection system misses the attack and fails to generate the intrusion alert. The very notion of priority based causality analysis comes into question here. Another key limitation is the fact that the output of PRIOTRACKER can be seen in Figure 8. However, there is no algorithm or description of the methods used to generate these reduced version of Forward Tracking Graph mentioned anywhere in the paper. This may lead to an assumption that these are all manually generated and separated by the dotted lines, which can be cumbersome for large attack scenarios. No efficient methods are described by which these reduced version of Forward Tracking Graph can be generated. And finally, the formula of reference score depends upon the occurrence of a repeated event on a host within a particular time period (one week here). This notion may not hold legit in some attack scenarios like WannaCry [23] that contains a series of initial attack steps which is very common in an enterprise network. In some instances of this attack, the victim unintentionally downloads a malicious .zip and runs it. Here unzipping this malicious file would not contribute to a different reference score since this process is often seen and thus lead to an incorrect rareness score.

V. NODOZE: Combatting Threat Alert Fatigue with Automated Provenance Triage

NODOZE [4] is an alert correlation system which mitigates *threat alert fatigue* issue that burdens cyber analysts with an overload of alarms generated by threat detection systems most of which tend to be benign or false alarms. It leverages the contextual and historical information from a generated alert and builds a causal dependency graph which then assigns an anomaly score to each edge and propagates it to the neighbouring edges using a novel network diffusion algorithm. The anomaly score is assigned based on the frequencies of all the events in an enterprise which are stored in an Event Frequency Database. Finally, a True Alert Dependency Graph is generated based on the *behavioural execution partitioning* which contains the most anomalous dependency paths generated from the candidate event.

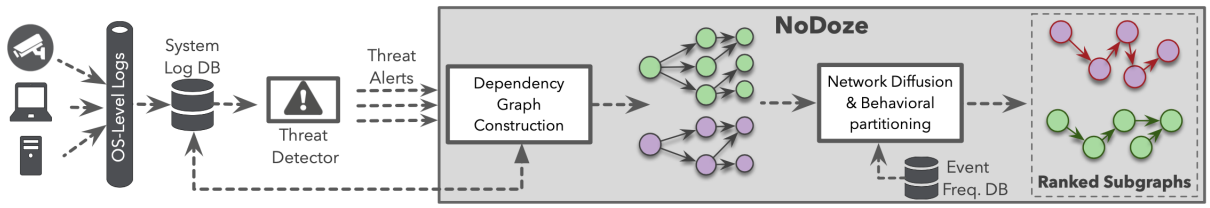

Fig. 9: NODOZE Architecture

A. *Design and System Overview*

NODOZE acts as an add-on to the existing intrusion detection systems, which aims to reduce false positives and generate a concise view of the generated threat alert. Similar to PRIOTRACKER [3], this alert correlation system also starts its workflow from an alert raised by the existing intrusion detection system and correlates alerts by assigning an anomaly score to each event in the provenance graph generated from the alert. Following are some definitions for the better understanding of the terms used in this paper: Similar to SLEUTH [2] subjects and objects correspond to processes and files, sockets respectively. Here a **dependency event** $\mathcal{E}$ is a 3-tuple relation $< SRC, DST, REL >$ where SRC is a process from where data flow is initiated, DST is a process or a object which receive the data flow, and REL represents the relationship between the process and the object. Following are the different dependency event relationships [Table VIII] modeled in NODOZE:

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

TABLE VIII: Dependency Event Relationships

A **dependency path** P is defined as the ordered sequence of the set of paths (of length n) which lead to a dependency event $\mathcal{E}_a$. It is divided into two types: A **control dependency path (CD)** of an event ε is a dependency path $P_{CD} = \{\varepsilon_1, \varepsilon_2, \cdots, \varepsilon_n\}$ such that REL is either *Pro_Start or Pro_End*. A **data dependency path (DD)** is a dependency path $P_{DD} = \{\varepsilon_1, \varepsilon_2, \cdots, \varepsilon_n\}$ such that the REL between the SRC and DST is anything but *Pro_Start or Pro_End*. A **true alert dependency graph** is the process of pruning of redundant nodes from the causal dependency graph by means of *behavioural execution partition* to generate a concise attack graph which contains the most anomalous dependency paths. Given a list of n alert events from the threat detector and user provided threshold parameters τ_l and τ_d, the key goal of NODOZE is to rank these alert events based on their anomaly score, prune out the false alarms by removing the alerts whose anomaly score is less than τ_d and

generate a true alert dependency graph with a maximum dependency path length of τ_l. The key intuition here is to calculate the anomaly score based on the whole dependency path of the candidate event. This algorithm generates the dependency paths of a candidate alert event and also calculates that path's anomaly score. In Algorithm 1, line 1 generates the complete dependency graph G for the alert event $\mathcal{E}_a$. All the dependency paths of the provided length τ_l for $\mathcal{E}_a$ are generated by a forward and a backward depth-first traversal and combined together. This is implemented in lines 2 to lines 6. In Algorithm 1, lines 7 to lines 10 then assigns an anomaly score to each event in the dependency path generated in the previous steps. First, a $N \times N$ transition probability matrix M is constructed for the given dependency graph G of the candidate alert event. N represents the total number of vertices in G and each entry in M_ε is calculated by:

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

This equation helps determine the frequency of a relationship between a particular source and destination. Further, a fanout score vector of each event in the dependency graph G is calculated by the means of IN and OUT scores which represents the degree of fanout from sender and receiver respectively (more on this later). Once the values in the transition probability matrix along with IN and OUT scores are calculated, a **Regularity Score** or a normal score of each dependency path is calculated. Given a dependency path $P = (\varepsilon_1, \cdots, \varepsilon_l)$ of length l, regularity score is defined as:

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

where IN and OUT are the sender and the receiver fanout score vectors. The right hand side of the equation measures the regularity or normality of an event ε in which SRC_i sends information to DST_i. This equation is normalized further to help generate a normalized anomaly score. Finally, the **Anomaly Score** (AS) is calculated as follows:

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

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.

1 G_α = GETDEPENDENCYGRAPH($\mathcal{E}_\alpha$)
2 $V_{src} \leftarrow$ GETSRCVERTEX($\mathcal{E}_\alpha$)
3 $V_{dst} \leftarrow$ GETDSTVERTEX($\mathcal{E}_\alpha$)
4 $L_b \leftarrow$ DFSTRAVERSALBACKWARD(G_α,V_{src},τ_l)
5 $L_f \leftarrow$ DFSTRAVERSALFORWARD(G_α,V_{dst},τ_l)
/* *Combine Backward and Forward Dependency Paths* */
6 $L_p \leftarrow$ COMBINEPATHS(L_b,L_f)
/* *Generate a transition matrix of an input graph using Eq. 1* */
7 M = GETTRANSITIONMATRIX(G)
8 **foreach** $P \in L_p$ **do**
/* *Calculate Path anomaly score using Eq. 2 and Eq. 3* */
9 $AS \leftarrow$ CALCULATESCORE(P,M)
/* *Append path and its anomaly score to a list* */
10 $L_{<P,AS>} \leftarrow L_{<P,AS>} \cup < P, AS >$
11 **end**
12 **return** $L_{<P,AS>}$

This equation ensures that if any path contains at least one abnormal event, it will be assigned a high anomaly score. The CALCULATESCORE in Algorithm 1 calculates the anomaly scores of the dependency paths of a candidate alert event. Since longer dependency paths would have higher anomaly scores compared to the shorter paths, a sampling-based approach is used to find the **decay factor** which normalizes the anomaly scores. In order to determine the decay factor α, a large sample of false alert events are taken, their dependency paths of different max lengths τ_l is constructed and their respective anomaly scores are calculated. Then, a key-value pair based map is used to store the path length and its corresponding average anomaly score. And finally, the ratio at which the anomaly score increases with increasing length from baseline length k is represented as the decay factor α. Equation 2 is finally re-calculated with this decay factor value which returns a normalized anomaly score for a given dependency path P of length l.

$$RS(P) = \prod_{i=1}^{l} IN(SRC_i) \times M(\varepsilon_i) \times OUT(DST_i) \times \alpha \qquad (4)$$

At the final stage, the **True Alert Dependency Graph** is generated by Algorithm 1. The inputs to this algorithm are the list of dependency path and anomaly score pairs and a merge threshold τ_m which quantifies the difference between the scores of benign and malicious paths. This algorithm merges high anomaly score paths until the difference is greater than the merge threshold. The value of τ_m is determined by using a training phase to calculate the average difference between benign and malicious paths. Finally, all the alerts are ranked based on their anomaly score and a cut-off threshold τ_d as mentioned earlier. If the anomaly score of an alert is greater than τ_d it is classified as a true alert and vice-versa. The threshold value is calculated from a training dataset consisting of true attacks and false alarms.

A very crucial data structure for the NODOZE architecture to work efficiently is the **Event Frequency Database**. This module counts the total number of occurrences of all events in an enterprise network and stores it in an external database. NODOZE queries this database to calculate scores for Equation 1. Further, this module needs to be updated periodically to update those counts. To calculate the frequencies of an event $\mathcal{E}_i =< SRC_i, DST_i, REL_i >$ following equations are used:

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

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

Here *hosts* are the number of hosts in the network, "$*$" is used to match any DST entity and *checkEvent* function returns the total number of occurrences of event $\mathcal{E}_i$ in host h in a pre-determined time window t. The number of occurrences of an event is counted in a very similar way to that of PRIOTRACKER [3] with the only difference here being the time windows which is 1 day here compared to a week in PRIOTRACKER.

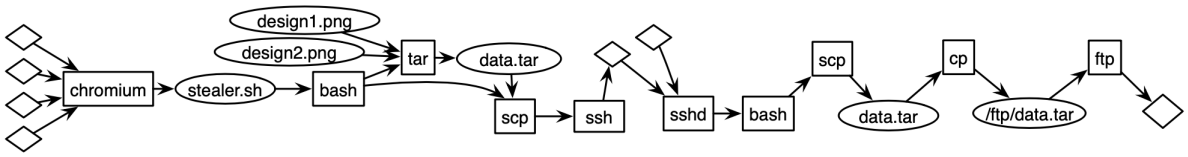

Fig. 10: True Alert Dependency Graph generated by NODOZE

B. *Evaluation*

The architecture of NODOZE and the Event Frequency Database is implemented in Java consisting of 9KLoC and 4KLoC respectively. The system audit logs are collected in PostgreSQL database using Windows ETW [24] and Linux Auditd [25] from 191 hosts (51 Linux and 140 Windows) for 5 days. A total of 50 attacks were simulated including 10 real-world APT attacks and 40 recent malwares from VirusTotal [26]. The baseline threat detection system used to generate the threat alert events is a commercial tool [27]. The results are summarised in Table IX. NODOZE was able to detect all the attacks in a very quick time period with a very negligible false positive rate. The evaluation dataset contains 400 GB of audit data with

Attack Case	**Baseline**				**NODOZE**							
	Dur(s)	**Ver**	**Edg**	**Size**	**Dur(s)**	**Ver**	**Edg**	**Size**	**CD(TP)**	**CD(FP)**	**DD(TP)**	**DD(FP)**
WannaCry	94	5948	8712	3320	18	19	21	49	100%	0%	100%	0.03%
Phishing Email	63	6002	148	3984	10	17	16	48	100%	0%	100%	0%
Data Theft	73	5364	23825	2208	41	23	24	65	100%	0%	100%	0%
Shellshock	31	2794	4031	3776	8	15	20	36	100%	0%	100%	0%
Netcat Backdoor	62	2914	6158	1968	14	12	11	48	88%	0%	84%	0%
Cheating Student	50	1217	22647	784	10	12	11	40	100%	0%	100%	0.07%
Passing the Hash	53	848	1026	560	11	8	8	36	100%	0%	100%	0%
wget-gcc	63	8323	8679	168	9	11	12	33	100%	0%	100%	0.01%
password-gzip-scp	68	8066	15318	5168	8	10	9	36	100%	0%	100%	0%
VPNFilter	20	2639	9774	1000	9	15	15	45	100%	0%	100%	0%

TABLE IX: Evaluation Results of NODOZE. Dur=Duration in seconds, Ver=Number of vertices, Edg=Number of Edges, Size in KB, CD=Control Dependency, DD=Data Dependency, TP=True Positive, FP=False Positive

approximately 1 billion OS-level log events. The Event Frequency Database was generated by consuming 10 days of daily OS-level system event in the enterprise and the underlying baseline TDS generated 364 threat alerts from all these simulated attacks. A true alert dependency graph generated from an attack is shown in Figure 10 which summarizes the attack steps in a very concise and simple graph representation. As it is evident from Table IX, NODOZE not only detects the attacks but does the same in a much less time than compared to the baseline provenance tracker. The number of edges and vertices in the true alert dependency graph is also much less compared to the baseline tracker. The runtime performance of NODOZE is highly impressive as 95% of all the alerts produced by the TDS are analyzed and responded by NODOZE in less than 40 seconds as shown in Table IX.

C. *Limitations*

The key contributions of NODOZE include the design and implementation of an automated platform-neutral alert correlation system for enterprise networks which leverages a novel network diffusion algorithm that propagates anomaly scores in dependency graphs generated from a candidate threat alert event and finally calculates the aggregate anomaly scores for threat alerts. Moreover, the true alert dependency graphs generated by NODOZE are concise and accelerates the investigation process without losing any important contextual attack information. NODOZE overcomes the dependence explosion problem of SLEUTH [2] by leveraging behavioural execution partition which effectively reduces the redundant nodes and produces a compact attack graph.

Focusing on the limitations of NODOZE, there are a few drawbacks in the technical details of the system. First, NODOZE would potentially miss the attacks in which attackers' steps are incredibly similar to those of benign programs. One of such

incidents happened in ccleaner [28] where attackers performed a typical APT-style attack and were able to inject a malware backdoor into the official version of the software from Avast. Since these malwares replicate those normal benign steps performed by the actual software (and some malicious attack steps), the flows manifested by this malicious ccleaner [29] [30] would be practically treated as benign by NODOZE leading to the attack not getting detected at all.

Second, NODOZE leverages an online database [31] to assign low IN and OUT scores to known malicious file extensions which are observed during audit data collection. However, with the advent of the several variations of the APT attacks, it is potentially unfeasible to rely on an online database that often depends on user submissions and can get delayed to update new malwares. Moreover, this website primarily focuses on windows' file extensions. Hence if NODOZE is implemented on a Linux/FreeBSD system, it can assign the default IN and OUT score of 0.5 to potentially malicious files which can eventually lead to a flawed anomaly score calculation. Further, the IN and OUT score assignment for socket connection is implemented by leveraging domain knowledge of the network. This leads to infeasibility for implementation purposes in case of a large enterprise network which consists of hundreds (if not thousands) of hosts.

Third, although the value of τ_l in Algorithm 1 is user provided, no particular intuition is explained how to determine this max path length threshold τ_l. Since the evaluation was based on simulated attacks, the authors knew what they were looking for and provided specific threshold values in different attack scenarios. However that might not be case in terms of adversarial testing cases, where analysts literally have no idea how long a dependency path can be and this is evident in the NetCat Backdoor experiment where the true positive (TP) rates for control and data dependency are 88% and 84% respectively.

And finally, the cut-off threshold τ_d used to determine if a candidate threat alert is a true attack or a false alarm plays a very crucial role in the efficacy of NODOZE. However, the paper provides a very vague one line explanation on how to calculate τ_d. *"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"*. Even in the attack scenarios explained, there is no mention of the anomaly scores of those alert event paths or how much the cut-off threshold value τ_d is.

VI. Discussion

Log based attack detection was first proposed by King *et al.* [21] which generated dependency graph based on OS-level system events in order to capture the prerequisites and consequences of an attack. Over the years, this has been finely tuned and improved by several papers. Tandon *et al.* [32] leveraged syscall arguments along with syscall sequences to detect malicious programs. Ben-Asher *et al.* [33] investigated the effects of knowledge to detect attacks. They established that contextual knowledge about alerts is more efficient in detection than compared to analysts' experience and prior knowledge. Milajerdi *et al.* [20] produced a strong signal for threat detection and correlated suspicious information flows by matching some pre-defined rules. In this technical critique report, I have summarized two different aspects of intrusion detection system- **threat detection** and **alert correlation**. Both of these aspects correspond to the four papers reviewed in this report. Each of these papers has a significant contribution to this area of intrusion detection systems. HERCULE[2] [1] is an automated multi-stage intrusion analysis system based on social networks. They define a long list of possible relationships between events. These relationships are leveraged to create graphs of events in which edges have a weight value, which is calculated using a quadratic optimization algorithm. The result returned by the system is a complete graph representing all the malicious events involved in a multi-step attack. SLEUTH [2] is a platform-neutral real-time attack detection and attack reconstruction system which leverages main memory dependency graph model and an efficient tag-based policy framework to produce attack scenarios. PRIOTRACKER [3] isn't directly aimed at constructing a compact scenario graph, but instead, it aims to include as much of attack activity as possible within a given amount of analysis time. NODOZE [4] prioritizes anomalous information flows to reduce the size of the graph generated by forward analysis. A key limitation to both of these systems is that since NODOZE and PRIOTRACKER rely on anomalous or rare events and/or anomalous information flows, sophisticated attackers can evade them by designing their malware to match the behaviors of benign applications. Moreover, these techniques require an external attack detector to initiate the analysis. Although the four papers cannot be compared directly to each other but a general comparison of the different aspects of these systems are summarized in Table X.

Kernel audit logs are a rich source of information for log-based causality analysis systems in order to detect attacks. There are a few headers for future research in threat detection and alert correlation aspects of an intrusion detection system. First, all these systems require audit logs to streamed into it or preserved in some storage for efficient threat detection. However, in an enterprise network, the number of hosts can easily range in hundreds (or even thousands) and the size of the audit logs just keep increasing with every passing day. If audit logs are stored in an enterprise, the total size of those files can rapidly escalate into terabytes in no time as hosts in an enterprise perform a huge number of operations on a daily basis. More research should be conducted on robust data retention techniques which would eliminate unnecessary logs in due time. Moreover, research should be performed in data reduction techniques so that the size of those log files do not exceed some certain sizes. All these would significantly decrease the storage overheads that exist in the present systems.

[2]HERCULE stands for Harmful Episode Reconstruction by Correlating Unsuspicious Logged Events, also as a tribute to Hercule Poirot, one of the most celebrated fictional detectives

	HERCULE	SLEUTH	PRIOTRACKER	NODOZE
Main Goals	Threat detection and attack scenario reconstruction	Threat detection and attack scenario reconstruction	Distinguish malicious attack events from benign events	Alert correlation using contextual information of threat alerts
Strengths	Detects threats and discovers attack communities embedded within graphs	Attack scenario reconstruction representing the root cause and impact analysis of the attack	Priority based causality tracker to detect abnormal causal dependencies	An alert correlation system which leverages a network diffusion algorithm in order to propagate and generate anomaly score for each alert
Weakness	Output graphs are difficult to analyze	Dependence explosion	No output graphs generated	Attacks that replicate benign application behaviour can remain undetected
Evaluation dataset	16 simulated attacks	73 million events	2.5 billion events	50 simulated attacks
Platform-neutral	Yes	Yes	Yes	Yes
Real-time detection	No	Yes	No	No
Implemented in	Python	C++ and Python	Java	Java

TABLE X: Comparison of different aspects of the four papers discussed in the report

Second, lateral movement is a key attack step in APTs as attackers tend to persist on an enterprise network by laterally moving from one host to another while undetected. In general, in order to detect such attack steps, a distributed framework is necessary instead of a per-host implementation. But it is very hard to achieve in an enterprise network which can consist of hundreds of hosts. Significant research should be conducted in that particular area.

REFERENCES

[1] K. Pei, Z. Gu, B. Saltaformaggio, S. Ma, F. Wang, Z. Zhang, L. Si, X. Zhang, and D. Xu, "Hercule: Attack story reconstruction via community discovery on correlated log graph," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, 2016, pp. 583–595.

[2] 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," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 487–504.

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

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

[5] "Mandiant: Exposing one of china's cyber espionage units," available at "https://www.fireeye.com/content/dam/fireeye-www/services/pdfs/mandiant-apt1-report.pdf".

[6] B. Barrett, "Equifax data breach," available at "https://www.wired.com/story/equifax-hack-china/".

[7] N. Ellen and H. Shane, "Dnc hack," available at "https://www.washingtonpost.com/world/national-security/how-the-russians-hacked-the-dnc-and-passed-its-emails-to-wikileaks/2018/07/13/af19a828-86c3-11e8-8553-a3ce89036c78_story.html".

[8] "Louvain algorithm- neo4j," available at "https://neo4j.com/docs/graph-algorithms/current/algorithms/louvain/".

[9] "Virustotal," available at "https://www.virustotal.com/gui/home/upload".

[10] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "Bitblaze: A new approach to computer security via binary analysis," in *International Conference on Information Systems Security*. Springer, 2008, pp. 1–25.

[11] K. Rieck, P. Trinius, C. Willems, and T. Holz, "Automatic analysis of malware behavior using machine learning," *Journal of Computer Security*, vol. 19, no. 4, pp. 639–668, 2011.

[12] V. Chipounov, V. Kuznetsov, and G. Candea, "S2e: A platform for in-vivo multi-path analysis of software systems," *ACM Sigplan Notices*, vol. 46, no. 3, pp. 265–278, 2011.

[13] T. K. Lengyel, S. Maresca, B. D. Payne, G. D. Webster, S. Vogl, and A. Kiayias, "Scalability, fidelity and stealth in the drakvuf dynamic malware analysis system," in *Proceedings of the 30th Annual Computer Security Applications Conference*, 2014, pp. 386–395.

[14] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools," *ACM computing surveys (CSUR)*, vol. 44, no. 2, pp. 1–42, 2008.

[15] "Neo4j graph platform," available at "https://neo4j.com/".

[16] "Titan, distributed graph database," available at "https://titan.thinkaurelius.com/".

[17] "Stinger," available at "http://www.stingergraph.com/".

[18] "Neteworkx," available at "https://networkx.github.io/documentation/stable/tutorial.html".

[19] M. N. Hossain, J. Wang, O. Weisse, R. Sekar, D. Genkin, B. He, S. D. Stoller, G. Fang, F. Piessens, E. Downing *et al.*, "Dependence-preserving data compaction for scalable forensic analysis," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 1723–1740.

[20] S. M. Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. Venkatakrishnan, "Holmes: real-time apt detection through correlation of suspicious information flows," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1137–1152.

[21] S. T. King and P. M. Chen, "Backtracking intrusions," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003, pp. 223–236.

[22] 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 the 2016 SIAM International Conference on Data Mining*. SIAM, 2016, pp. 72–80.

[23] "Wannacry ransomware," available at "https://www.fireeye.com/blog/products-and-services/2017/05/wannacry-ransomware-campaign.html".

[24] "Event tracing for windows," available at "https://docs.microsoft.com/en-us/windows/win32/etw/event-tracing-portal".

[25] "Auditd for linux," available at "https://linux.die.net/man/8/auditd".

[26] "Virusshare," available at "https://virusshare.com/".

[27] "Automated security intelligence (asi)," available at "https://www.nec.com/en/global/techrep/journal/g16/n01/160110.html".

[28] Wikipedia, "Ccleaner," available at "https://en.wikipedia.org/wiki/CCleaner".

[29] T. Warren, "Hackers hid malware in ccleaner software," 2017, available at "https://www.theverge.com/2017/9/18/16325202/ccleaner-hack-malware-security".

[30] L. H. Newman, “Inside the unnerving supply chain attack that corrupted ccleaner,” 2017, available at ”https://www.wired.com/story/inside-the-unnerving-supply-chain-attack-that-corrupted-ccleaner/”.
[31] “File-extensions,” available at ”https://www.file-extensions.org/filetype/extension/name/dangerous-malicious-files”.
[32] G. Tandon and P. K. Chan, “On the learning of system call attributes for host-based anomaly detection,” *International Journal on Artificial Intelligence Tools*, vol. 15, no. 06, pp. 875–892, 2006.
[33] N. Ben-Asher and C. Gonzalez, “Effects of cyber security knowledge on attack detection,” *Computers in Human Behavior*, vol. 48, pp. 51–61, 2015.