INVESTIGATING SYSTEM INTRUSIONS WITH DATA PROVENANCE ANALYTICS

BY

WAJIH UL HASSAN

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2021

Urbana, Illinois

Doctoral Committee:

Professor Adam Bates, Chair
Professor Michael Bailey
Professor Carl Gunter
Professor Vern Paxson, University of California, Berkeley
Professor Dongyan Xu, Purdue University

ABSTRACT

To aid threat detection and investigation, enterprises are increasingly relying on commercially available security solutions, such as Intrusion Detection Systems (IDS) and Endpoint Detection and Response (EDR) tools. These security solutions first collect and analyze audit logs throughout the enterprise and then generate threat alerts when suspicious activities occur. Later, security analysts investigate those threat alerts to separate false alarms from true attacks by extracting contextual history from the audit logs, i.e., the trail of events that caused the threat alert.

Unfortunately, investigating threats in enterprises is a notoriously difficult task, even for expert analysts, due to two main challenges. First, existing enterprise security solutions are optimized to miss as few threats as possible – as a result, they generate an overwhelming volume of false alerts, creating a backlog of investigation tasks. Second, modern computing systems are operationally complex that produce an enormous volume of audit logs per day, making it difficult to correlate events for threats that span across multiple processes, applications, and hosts.

In this dissertation, I propose leveraging data provenance analytics to address the challenges mentioned above. I present five provenance-based techniques that enable system defenders to effectively and efficiently investigate malicious behaviors in enterprise settings. First, I present NoDoze, an alert triage system that automatically prioritizes generated alerts based on their anomalous contextual history. Following that, RapSheet brings benefits of data provenance to commercial EDR tools and provides compact visualization of multi-stage attacks to system defenders. Swift then realized a provenance graph database that generates contextual history around generated alerts in real-time even when analyzing audit logs containing tens of millions of events. Finally, OmegaLog and Zeek Agent introduced the vision of universal provenance analysis, which unifies all forensically relevant provenance information on the system regardless of their layer of origin, improving investigation capabilities.

To my mother, Fehmida Zahid.

ACKNOWLEDGMENTS

During my Ph.D. journey, I have met many amazing people to whom I am forever grateful. First and foremost, I would like to thank my advisor, Professor Adam Bates, for his guidance, advice, and support toward my success. Adam has shown me the type of professor, advisor, and mentor that I aspire to be. None of my achievements would have been possible without Adam's help and patience. I will always remain thankful to him for believing in me and providing me with the opportunity to conduct research in this field.

I would also like to express my profound gratitude to my committee members: Professor Michael Bailey, Professor Carl Gunter, Professor Vern Paxson, and Professor Dongyan Xu. They provided invaluable insights, feedback, and suggestions for my dissertation while helping me better understand how to achieve my career goals.

This dissertation benefited from my collaborations with the researchers from NEC Laboratories America. I would like to thank them for giving me a chance to work at NEC Laboratories America on the NoDoze and Swift projects. I also thank my other research collaborators during my Ph.D., including Dr. Daniel Marino, Dr. Darko Marinov, Dr. Owolabi Legunsen, Dr. Tianyin Xu, Dr. Thomas Moyer, Dr. Mohammad Noureddine, Dr. Fareed Zaffar, and Dr. Rashid Tahir.

Many thanks to my amazing lab mates in the Secure and Transparent Systems (STS) Lab: Pubali Datta, Saad Hussain, Jay Pandey, Riccardo Paccagnella, and Akul Goyal. They made this Ph.D. experience even more enriching and enjoyable. I also thank my other colleagues at the University of Illinois at Urbana-Champaign (UIUC), especially Wing Lam, Angello Astorga, Tifany Yung, Farah Hariri, Xinyue Xu, Avesta Hojjati, Güliz Seray Tuncay, Qi Wang, August Shi, Alex Gyori, Zane Ma, Deepak Kumar, Muhammad Samir Khan, Muhammad Adil Inam, Muhammad Haris Mughees, Abdulrahman Mahmoud, and Benjamin Ujcich.

I feel privileged to have had amazing and caring friends who made Urbana-Champaign my home away from home. I would like to thank Hassan Shahid for being an awesome roommate; Muhammad Huzaifa, for planning and managing our memorable trips into the mountains; Shalan Naqvi, for your witty humor; Gohar Irfan, for constantly nagging me to make biryani over the weekends; Saad Hussain, for doing absolutely nothing; Hashim Sharif, for encouraging me to apply to UIUC; and Zainab Rahil, for all the laughs and UIUC-related gossips. I cannot imagine how my graduate life would have been if I did not share all these good memories with them. Finally, I would like to thank Tooba Shoaib for inviting me to

countless dinners, for always giving helpful advice, and for ensuring that my memories of graduate school would be overwhelmed by happy moments.

Above all, I would like to express my deepest appreciation to my family for their endless and unconditional love and support. I dedicate this dissertation to my mother, Fehmida Zahid, who always encouraged my sense of curiosity and fostered my love of learning from a very early age. Thank you for all the sacrifices you have made for me and for doing everything possible to help me grow and excel in life.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

System intrusions are becoming progressively more subtle and sophisticated. Using an approach exemplified by the "living-off-the-land" attack strategy of *Advanced Persistent Threats* (APTs), attackers now lurk in target systems for extended periods to expand their reach before initiating devastating attacks. By avoiding actions that would immediately arouse suspicion, attackers can achieve dwell times that range from weeks to months, as was the case in numerous high-profile data breaches, including Target [1], Equifax [2], and SolarWinds [3].

To combat these threats, enterprises are deploying threat detection and investigation systems, such as Intrusion Detection System (IDS) and Endpoint Detection and Response (EDR). These systems constantly monitor enterprise-wide activities and generate a threat alert if a suspicious activity happens. Security analysts then manually sift through these alerts to find a signal that indicates a true attack. Once a true attack is identified, analysts perform an incident response and recovery process. Depending upon the volume of alerts and the analysis tools available to the analyst, this threat investigation process can typically range from hours to days for an individual threat alert [4].

While threat detection and investigation systems are vital for enterprise security, two challenges undermine their usefulness in practice. The first challenge is that the rule-set used to detect threats is optimized for recall, not precision; that is, rule-set curators attempt to describe all procedures that have any possibility of being attack related, even if the same procedures are widely employed for innocuous purposes. As a result, current enterprise security solutions generate an overwhelming volume of false alerts [5, 6, 7, 8], creating a backlog of investigation tasks. In fact, these systems are one of the key perpetrators of the "threat alert fatigue" problem – a phenomenon in which analysts do not respond, or respond inadequately, to alerts because they receive so many each day.

The second challenge comes dubious nature of generated threat alerts. After receiving an alert, the first job of an analyst is to determine the alert's veracity. For such validation, analysts review the context around the triggered alert by leveraging audit logs. However, modern computing systems are operationally complex that generate an enormous volume of audit logs per day, making it difficult to recover and correlate events for threats span across multiple processes, applications, hosts, and networks. Security Indicator & Event Management (SIEM) products are often the interface through which this task is performed (e.g., Splunk [9]), allowing analysts to write long ad-hoc queries to join attack stages, provided that they have the experience and expertise to do so.

In this dissertation, I use principled approaches based on data provenance to solve the above-mentioned challenges. Data provenance can be applied to audit logs to parse host events into provenance graphs that describe the totality of system execution and facilitate causal analysis of system activities. In recent years, significant advancements have been made to improve the fidelity [10, 11, 12, 13, 14, 15, 16, 17] and efficiency [18, 19, 20, 21, 22, 23, 24, 25] of data provenance analysis and recent research indicate that provenance analysis can even be used to network debugging [26, 27], access control [28], and intrusion detection [29]. However, leveraging data provenance to aid the investigation of system intrusions in enterprise settings remained an open research challenge.

1.1 DISSERTATION STATEMENT

My research builds practical solutions for securing complex networked computer systems by leveraging data provenance analytics. In this dissertation, I will demonstrate that by incorporating data provenance in enterprise security solutions, system defenders can generate contextual history of system execution from enormous audit logs. This dissertation proposes using this contextual history to identify the root cause and impact of system intrusions, triage threat alerts, summarize long-lived attack campaigns, and correlate the audit logs collected from the system, application, and network layers.

The central thesis of this dissertation is as follows: *understanding of contextual history of system execution from a diverse range of vantage points enables more effective and efficient investigation of intrusions as compared to traditional solutions.*

1.2 SUMMARY OF CONTRIBUTIONS

In this dissertation, I designed five scalable systems that bring benefits of data provenance to the enterprise security ecosystem. Below I highlight key ideas and contributions of these systems:

1.2.1 NoDoze

As I described earlier, in practice, there are more alerts than analysts can properly investigate, creating a "threat alert fatigue" where analysts miss true attack alerts in the noise of false alarms. To address this limitation, I will present NoDoze [30], an alert triage system that automatically prioritized generated alerts based on their anomalous contextual history. My key insight was that each event's suspiciousness in the provenance graph should

be adjusted based on the suspiciousness of neighboring events in the graph. To this end, I assigned an anomaly score to each edge in the provenance graph based on the frequency with which related events have happened before in the organization. After that, I used a novel adaptation of network diffusion algorithm to propagate and aggregate those anomaly scores along the neighboring edges of the provenance graph. Finally, I used those aggregate anomaly scores to triage alerts. To show the merits of my approach, I deployed and evaluated NoDoze at NEC Labs America using 364 alerts generated by NECs commercial threat detector, and discovered that NoDoze consistently ranked the true alerts higher than the false alerts. Notably, I found that my technique not only could prioritize alerts but could also reduce false alarms by 84% and saved analysts more than 90 hours of investigation time per week.

1.2.2 RapSheet

NoDoze showed the efficacy of triaging alerts based on anomalous contextual history; however, NoDoze is only applicable if anomalous events exist in the provenance graph. In stealthy attacks, the attackers often avoid using any anomalous activities. To detect such attacks, organizations deploy Endpoint Detection and Response (EDR) systems, which match audit logs against a knowledge base of adversarial behaviors. However, during my investigation, I found that EDR systems not only generate a high number of false alarms but also only keep audit logs for short periods due to the huge resource burden of log retention. In practice, EDR systems often delete audit logs before an attack investigation is ever initiated. To solve those limitations, I will present a provenance-based EDR called RapSheet [31]. RapSheet leveraged the notion of the *tactical provenance graphs* that, rather than encoding low-level system event dependencies, reasoned about causal dependencies between alerts. I designed a novel threat scoring scheme that assesses each alert's severity based on its tactical provenance graph, enabling effective triage of EDR-generated alerts. Moreover, to provide long-term log retention, I demonstrated that instead of storing unwieldy low-level audit logs, maintaining a "skeleton graph" is minimally sufficient for forensic analysis. This skeleton graph retained just enough context to not only identify causal links between the existing alerts but also any alerts that may be triggered in the future. In evaluating RapSheet, I considered 55,000 alerts generated by Symantec Enterprise machines, and discovered that my approach consistently ranked truly malicious alerts higher than false alarms. Moreover, I found that my skeleton graph approach could reduce the long-term burden of log retention by up to 87%.

1.2.3 Swift

A notable unaddressed limitation of NoDoze and RapSheet was that provenance graph construction took upwards of hours to respond to investigator queries, creating a greater opportunity for attackers to extend their reach and dwell longer on the victims' machine. To limit the attacks exposure and allow for faster post-breach threat hunting, I will present a high-throughput and low-latency provenance graph generation framework called Swift [32]. Swift consisted of an in-memory graph database that enabled space-efficient graph storage and online causality tracking with minimal disk operations. I developed a hierarchical storage system that only kept forensically relevant parts of the provenance graph in the main memory and spilled the rest to the disk. To identify graph elements that were likely to be relevant during forensic investigations, I designed an asynchronous cache eviction policy that calculates the most suspicious part of the provenance graph and caches only that part in the main memory. Through extensive evaluation, I demonstrated that Swift could generate provenance graphs of 10 real-world APT attacks from a database of over 300 million events in less than 2 minutes, 10,000x faster than state-of-the-art solutions.

1.2.4 OmegaLog

While Swift paved the path for scalable auditing, through further investigation, I realized that existing auditing frameworks based on system-layer events (e.g., syscalls) often generate inaccurate causal analysis results due to dependency explosion: a problem in which an output of a long-running process is falsely assumed to be causally dependent on all the preceding inputs. Dependency explosion happens in large part because of the semantic gap between system-layer events and application-layer behaviors. To solve this limitation, I will present OmegaLog [33]. I will show that application-layer semantics are usually found in applications' innate event logs, and through integrating such application logs into system logs, we can solve the dependency explosion problem. To that end, I first modeled application logging behaviors using static binary analysis. I demonstrated that those models could be used to reconcile application-layer events with system-layer accesses and partition a long-running process into semantically independent execution units. During the evaluation, I undertook an expensive series of attack case studies and discovered that my approach enabled semantically rich and accurate attack reconstructions with just 4% runtime overhead. Most excitingly, I demonstrated that, unlike existing solutions, my approach did not require any invasive instrumentation to solve the dependency explosion problem.

1.2.5 Zeek Agent

Enterprise monitoring tools collect various types of logs, such as system logs and network logs, to help analysts spot possible intrusions. However, the monitoring tools commonly used today collect network and host logs in a siloed manner, making it extremely difficult for security analysts to correlate events across boundaries of these logs when investigating system intrusions. To address this challenge and enable cross-log causal analysis, we designed an open source endpoint monitoring system called Zeek Agent [34]. Zeek Agent transparently collects host monitoring logs and seamlessly correlates those host logs with the network flows present in Zeek logs [35]. Moreover, Zeek Agent allows security analysts to construct a unified provenance graph from the correlated host and network logs to accelerate the threat investigation process. Evaluation results show that Zeek Agent is scalable, extensible, and enables cross-log analysis between host and network logs, with low execution overhead.

This dissertation is organized as follows:

- Chapter 1 has introduced the challenges and benefits of using data provenance in the enterprise security domain. I have highlighted my main contributions to this field of research and discussed different tools that I developed over the course of my Ph.D. to improve the threat investigation capabilities of enterprise security solutions.

- Chapter 2 discusses the challenge of "threat alert fatigue" problem. To tackle this challenge, I will present NoDoze, an automatic alert triage and investigation system based on provenance graph analysis. I will discuss my novel network diffusion algorithm that assigns anomaly scores for triaging threat alerts. I will also discuss how I deployed and evaluated NoDoze at NEC Labs America.

- Chapter 3 explains in detail the motivation, design, implementation, and evaluation of RapSheet. First, I will describe the enterprise security solution landscape. Then I will discuss how I incorporated data provenance in Symantec's enterprise security solution to make it more practical.

- Chapter 4 presents Swift, a threat investigation system that provides high-throughput causality tracking and real-time causal graph generation capabilities. I will discuss how we designed an in-memory graph database that enables online causality tracking with minimal disk operations. I will also describe a hierarchical storage system that keeps forensically-relevant part of the causal graph in the main memory while evicting the rest to the disk.

- Chapter 5 details OmegaLog, an end-to-end provenance tracker that merges application event logs with the system log to generate a *universal provenance graph*. This graph combines the causal reasoning strengths of whole-system logging with the rich semantic context of application event logs, allowing investigators to reason more precisely about the nature of attacks.
- Chapter 6 discusses the design and implementation of Zeek Agent framework that transparently collects host monitoring logs and combines them with Zeek network logs. I will discuss unique challenges in correlating these siloed log streams and Zeek Agent's technique to tackle those challenges.
- Chapter 7 outlines related work and their limitations which serves as a motivation for this dissertation.
- Chapter 8 concludes this dissertation.

CHAPTER 2: NODOZE: PROVENANCE-BASED THREAT ALERT TRIAGE

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.

Unfortunately, these automated systems are notorious for generating high rates of false alarms [5, 6, 7]. 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 [8]. 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 [1], 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 Figure 2.1, we demonstrate the growth of alerts generated by a commercial TDS [36] at NEC Labs America comprising 191 hosts.

The threat alert fatigue problem is, at least partially, caused by the fact that existing academic [37, 38] and commercial [39, 40] 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 [41]. Even though contextual alerting has proven to be most effective in the alert triage process [42], 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]

Data provenance analysis [10, 43] 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)

[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.

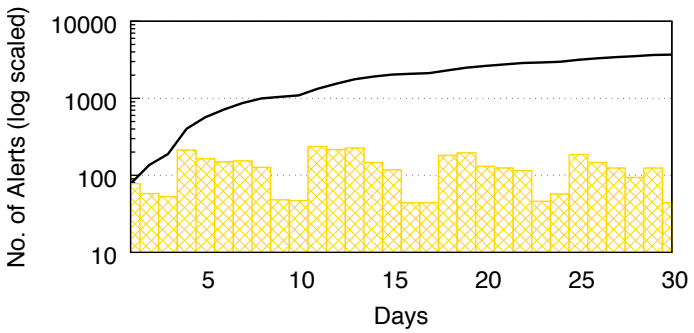


Figure 2.1: Growth of alerts in an enterprise during a given month.

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 [13]. 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 [18], making the diagnosis of attacks prohibitively difficult.

In this chapter, 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.

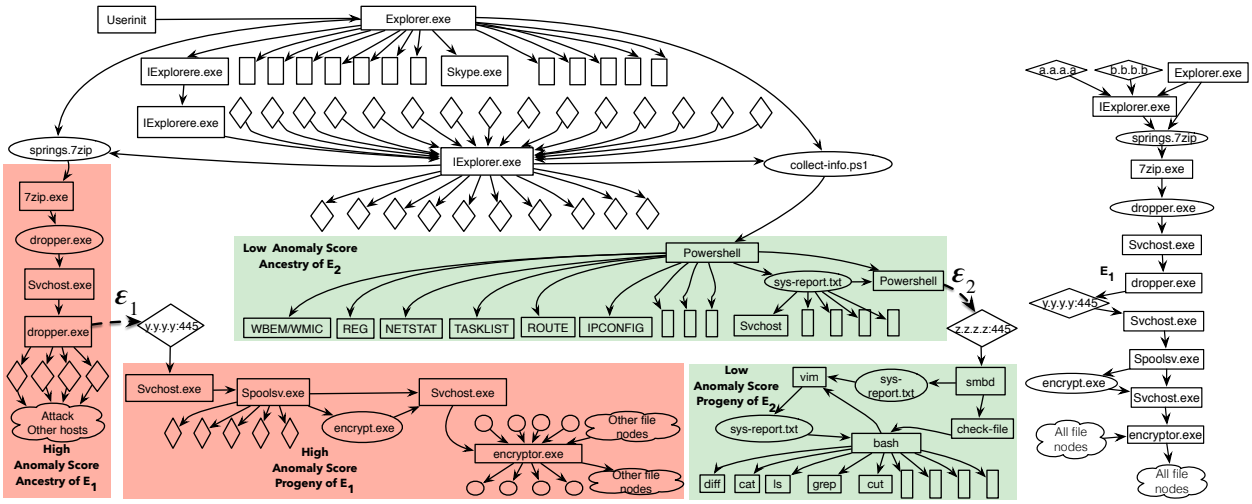
2.1 MOTIVATION

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 [44] 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 Section 2.4.

2.1.1 Motivating Attack Example

WannaCry ransomware is a popular attack that affected around 0.2 million systems across 150 countries in May 2017 [45]. It is essentially a cryptoworm which targets computers running the Microsoft Windows OS with vulnerable EternalBlue [46]. 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



(a) (b)

Figure 2.2: WannaCry attack scenario described in 2.1.1. **(a)** Part of the threat alerts' dependency graph generated by prior approaches [10, 43]. Some edges have been omitted for clarity. **(b)** Concise dependency graph generated by NoDoze.

technical 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 [10, 43] 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. Figure 2.2a shows the simplified dependency graph generated by existing tools for alert events $\mathcal{E}_1$ and $\mathcal{E}_2$.

2.1.2 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 [5, 6, 7, 47, 48]. 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 [8]. Many heuristic- and rule-based static approaches have been proposed to mitigate this problem [49, 50, 51, 52]. 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 Figure 2.2a. 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 Figure 2.2a. 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.

Prior solutions to the dependency explosion problem [12, 13, 16, 21] 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. Fur-

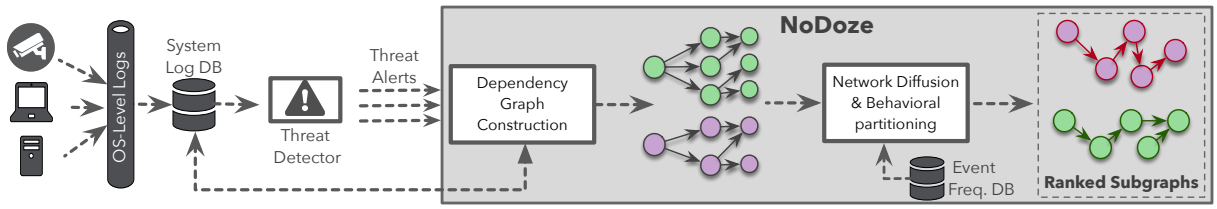


Figure 2.3: Overview of NoDoze. Alerts generated by threat detector are provided to NoDoze, which ranks the alerts based on their aggregate anomaly scores and produces concise alert dependency graphs for investigation.

thermore, 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.

2.2 DESIGN OVERVIEW

The overall workflow of NoDoze system to triage alerts based on anomaly scores is shown in Figure 2.3. 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 [12, 13, 16, 21] 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.

Figure 2.2b 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 Figure 2.2a 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 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 Figure 2.2b. Observe that in Figure 2.2a, `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` and `b.b.b.b` since these have higher anomaly scores than the other normal socket connections.

2.3 ALGORITHM

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

2.3.1 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 2.1. For example, in Figure 2.2a a dependency event $\mathcal{E}_1$ is represented as <`dropper.exe`, `y.y.y.y:445`, `IP_Write`>.

Table 2.1: Dependency Event Relationships

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

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 Figure 2.4.

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 Figure 2.2a.

True Alert Dependency Graph. As we discussed in Section 2.1, due to long running programs there are false dependency events in the dependency graph. Due to false depen-

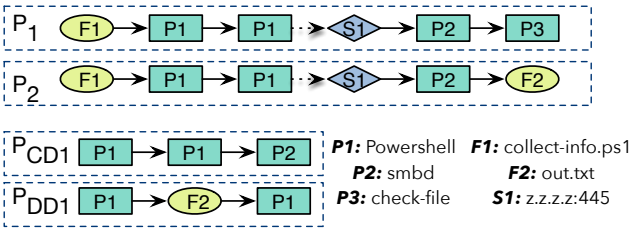


Figure 2.4: Example dependency paths of length 5 for alert event $\mathcal{E}_2$ from the motivating example (Section 2.1).

dencies, 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.

2.3.2 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 (Section 2.1), 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.

2.3.3 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 2.1, 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 2.1, 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:

$$M_\varepsilon = probability(\varepsilon) = \frac{|Freq(\varepsilon)|}{|Freq_{src\_rel}(\varepsilon)|} \tag{2.1}$$

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 Section 2.4.1.

Let's consider an alert event $\mathcal{E}_1$:=<`dropper.exe`, `y.y.y.y:445`, `IP_Write`> from Figure 2.2a. We first calculate $Freq(\mathcal{E}_1)$ by counting the number of events that have happened in our frequency event database where *src* ∈ `dropper.exe` , *dst* ∈ `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* ∈ `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 Section 2.4.1.

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 (Section 2.1), 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 Section 2.3.4.

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:

$$RS(P) = \prod_{i=1}^{l} IN(SRC_i) \times M(\varepsilon_i) \times OUT(DST_i) \tag{2.2}$$

where *IN* and *OUT* are the sender and receiver vectors, and M is calculated by Equation 2.1. In Equation 2.2, $IN(SRC_i) \times M(\varepsilon_i) \times OUT(DST_i)$ 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:

$$AS(P) = 1 - RS(P) \tag{2.3}$$

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 2.1, function CalculateScore generates anomaly scores of given dependency paths.

2.3.4 IN and OUT Scores Calculation

As mentioned above, Equation 2.2 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

Algorithm 2.1: 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. 2.1 */*
M = GETTRANSITIONMATRIX(G)
foreach $P \in L_p$ **do**
  / Calculate Path anomaly score using Eq. 2.2 and Eq. 2.3 */*
  $AS \leftarrow$ CALCULATESCORE(P,M)
  / Append path and its anomaly score to a list */*
  $L_{<P,AS>} \leftarrow L_{<P,AS>} \cup < P, AS >$
end
return $L_{<P,AS>}$

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 2.4 and Equation 2.5 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.

$$IN(v) = \frac{|T'_{to}|}{|T|} \tag{2.4}$$

$$OUT(v) = \frac{|T'_{from}|}{|T|} \tag{2.5}$$

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 [19]. 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 [53] 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 [54] address to assign low *IN* and *OUT* score.

2.3.5 Anomaly Score Normalization

For each alert causal path P, we calculate the anomaly score using Eq. 2.2 and Eq. 2.3. 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 2.2. 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 shown in Algorithm 2.2. Once the decay factor is calculated, the regularity score Equation 2.2 becomes as follows:

$$RS(P) = \prod_{i=1}^{l} IN(SRC_i) \times M(\varepsilon_i) \times OUT(DST_i) \times \alpha \tag{2.6}$$

2.3.6 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

Algorithm 2.2: CalculateDecayFactor

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

Output: Decay Factor α

```
1 M = KeyValue Store of Path Length and Avg. Anomaly Score
2 foreach E ∈ L_E do
3     for i ← 0 to τ_l do
          /* Use Algorithm 2.1 to generate anomaly score for given event and max path length */
4         L_<P,AS> = GetPathAnomalyScore(E,i)
          /* Takes the average of anomaly scores for each path length and store in map */
5         M[i] ← AverageScore(L_<P,AS>,M[i])
6     end
7 end
   /* Returns the ratio at which score increases with length from the baseline */
8 α ← GetDecayFromBaseline(M,k)
9 return α
```

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 2.3 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.

Algorithm 2.3: 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 ← 0 to sizeof(L_PS) − 1 do
    /* Path and its anomaly score pair */
    < P_1, S_1 > ← L_PS[i]
    < P_2, S_2 > ← L_PS[i + 1]
    if S_1 − S_2 < τ_m then
        G ← G ∪ P_1
        G ← G ∪ P_2
    end
end
return G
```

2.3.7 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.

2.3.8 Time Complexity of our Algorithm

The dependency paths search for an alert event is done using depth-first search (DFS) traversal with bounded depth D. 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 2.2 runs for each path so time complexity is $\mathcal{O}(|PD|)$ where P is the total number of dependency paths for the alert event.

2.4 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?

RQ2 How much can NoDoze reduce the dependency graph of a true alert without sacrificing the vital information needed for investigation?

RQ3 How much of investigator's time can NoDoze save when used in an enterprise setting?

RQ4 What is the runtime overhead of NoDoze?

2.4.1 Implementation

We implement NoDoze for an enterprise environment. We collected system event logs in PostgreSQL database using Windows ETW [55] and Linux Auditd [56]. In order to calculate the transition probability matrix M, IN score vector, and OUT score vector for Equation 2.2, 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 [18, 20] 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. 2.1) are as follows:

$$Freq(\mathcal{E}_i) = \sum_{h}^{hosts} checkEvent(SRC_i, DST_i, REL_i, h, t) \tag{2.7}$$

$$Freq_{src\_rel}(\mathcal{E}_i) = \sum_{h}^{hosts} checkEvent(SRC_i, *, REL_i, h, t) \tag{2.8}$$

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 [57]. Note that in our experiments t is set to stable window size (discussed in Section 2.3.4) which is 1 day. Finally, in Eq. 2.8 "$*$" means any DST entity.

2.4.2 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 [58]. A short description of each APT attack with generated threat alert is shown in Table 2.2.

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 [36] 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.

2.4.3 Baseline TDS

The baseline TDS we used to generate threat event alerts is a commercial tool [36]. Details regarding anomaly detection models used in this tool can be found here [64]. 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 2.2: Real-world attack scenarios with short descriptions and generated threat alerts by underlying TDS.

Attacks	**Short Description**	**True Threat Alert**
WannaCry [44]	Motivating example discussed in Section 2.1	See Section 2.1
Phishing Email [59]	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 [20]	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 [60]	An attacker utilized an Apache server to trigger the Shellshock vulnerability in Bash multiple times.	<bash, nc.traditional, Pro_Start>
Netcat Backdoor [61]	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 [21]	A student downloaded midterm scores from Apache and uploaded a modified version.	<Apache2, /www/newscores, File_Write>
Passing the Hash [62]	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 [22]	Malicious source files were downloaded and then compiled.	<wget, x.x.x.x:80, IP_Read>
passwd-gzip-scp [22]	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 [63]	An attacker used known vulnerabilities [60] 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>

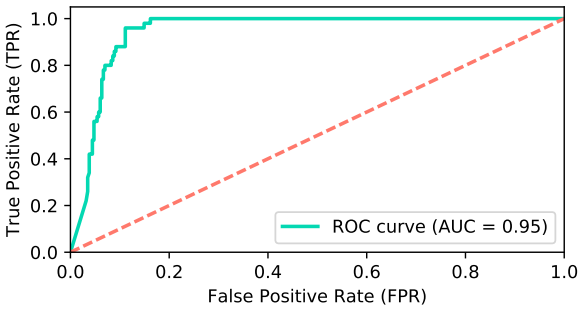


Figure 2.5: ROC curve for our experiments using NoDoze along with TDS.

2.4.4 Improvement Over Existing TDS

The first research question of our evaluation is how much NoDoze improves the accuracy of existing TDS [37, 38, 64, 65] which are based on heuristics and single event matching rules. To answer this question, we used NoDoze along with the baseline TDS [36]. 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 Figure 2.5. 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. Figure 2.6 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.

2.4.5 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

Table 2.3: Comparison of dependency graphs generated by NoDoze against prior tools [10, 43]. 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%

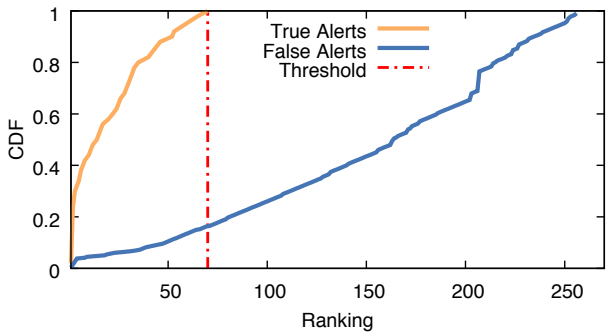


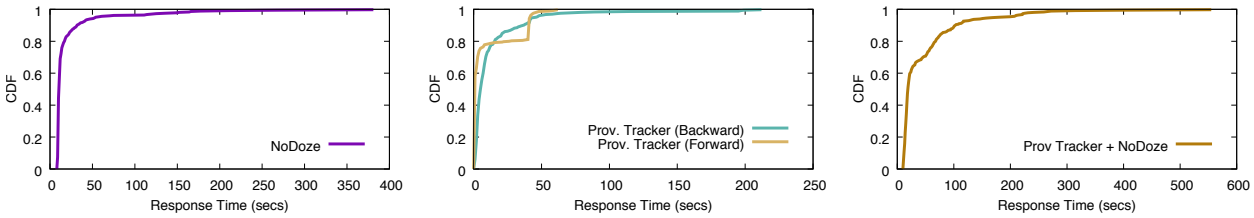
Figure 2.6: CDF for ranked true and false alerts based on aggregate anomaly score.

in Table 2.3. 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.

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 Section 2.3.1) with their true positive (TP) and false positive (FP) rates. Intuitively, the TP means the number of truly attack related edges present 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 2.2 in Section 2.5.



(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.

Figure 2.7: CDF of query response times for all the 364 threat alerts in our dataset.

2.4.6 Time Saved Using NoDoze

Recent studies [5, 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.

2.4.7 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 Figure 2.7a. 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 Figure 2.7b. 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 [20, 66] 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 Figure 2.7c. 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 parallelized easily using existing distributed graph processing frameworks.

2.5 CASE STUDIES

2.5.1 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 Figure 2.8a 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

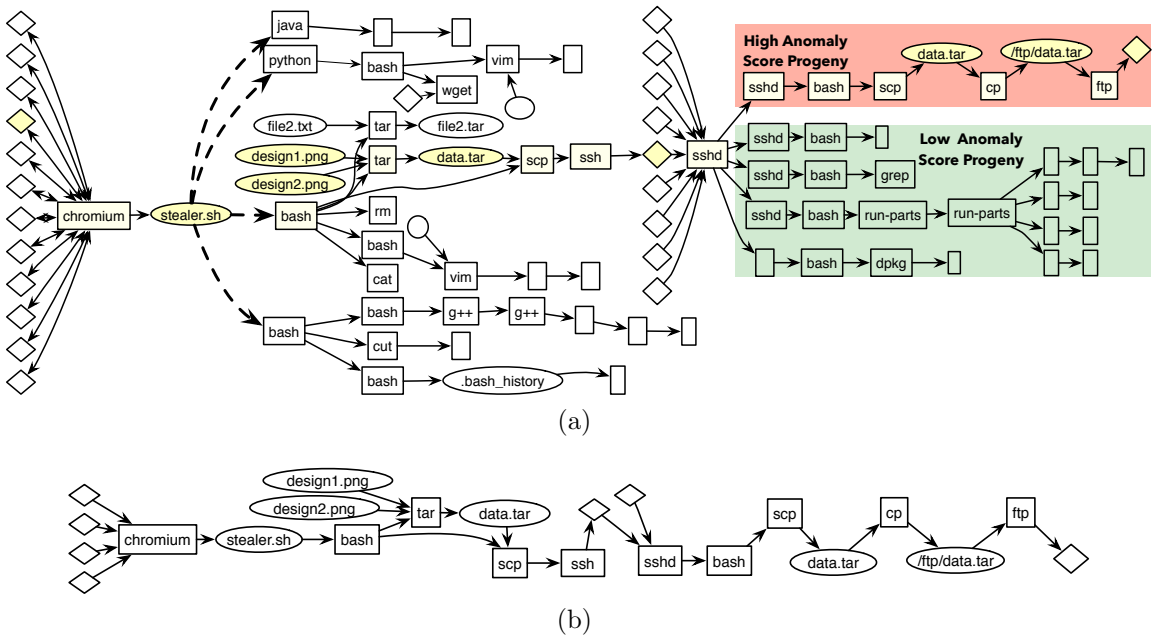


Figure 2.8: Data theft attack scenario discussed in Section 2.5.1. **(a)** Part of dependency graph generated by traditional tools. **(b)** Concise true alert dependency graph generated by NoDoze.

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 Figure 2.8b. Observe that in Figure 2.8a, 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 2.3 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.

2.5.2 ShellShock Attack

Scenario. In this attack scenario, attacker targets a ShellShock vulnerable Apache web-server 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`,

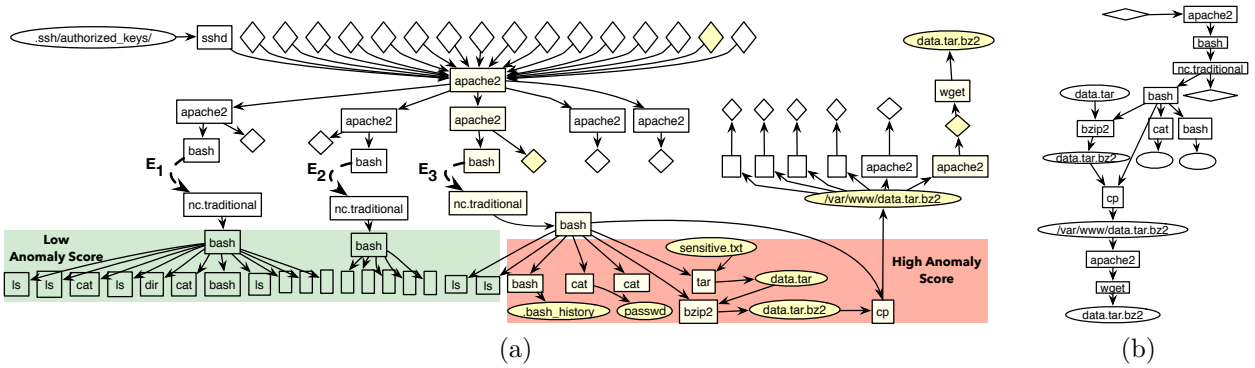


Figure 2.9: ShellShock attack scenario discussed in Section2.5.2. **(a)** Part of dependency graph generated by traditional tools. **(b)** Concise true alert dependency graph generated by NoDoze.

`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 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 Figure 2.9a.

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.

Figure 2.9b 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.

2.6 DISCUSSION & LIMITATIONS

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.*, [49, 52]).

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 [67]. 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 [36] 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 [64] and is similar to existing commercial and academic TDS [37, 38, 39, 40, 48] in FPR.

CHAPTER 3: RAPSHEET: BRINGING DATA PROVENANCE TO COMMERICAL SECURITY SOLUTIONS

The canonical enterprise solution for combatting APTs is known as *Endpoint Detection and Response (EDR)*. EDR tools constantly monitor activities on end hosts and raise threat alerts if potentially-malicious behaviors are observed. In contrast to signature scanning or anomaly detection techniques, EDR tools hunt threats by matching system events against a knowledge base of adversarial Tactics, Techniques, and Procedures (TTPs) [68], which are manually-crafted expert rules that describe low-level attack patterns. TTPs are hierarchical, with tactics describing "why" an attacker performs a given action while techniques and procedures describe "how" the action is performed. According to a recent survey, 61% of organizations deploy EDR tools primarily to provide deep visibility into attacker TTPs and facilitate threat investigation [69]. MITRE's ATT&CK [70] is a publicly-available TTP knowledge base which is curated by domain experts based on the analysis of real-world APT attacks, and is one of the most widely used collections of TTPs [71, 72, 73]. In fact, all 10 of the top EDR tools surveyed by Gartner leverage the MITRE ATT&CK knowledge base to detect adversary behavior [74].

While EDR tools are vital for enterprise security, three challenges undermine their usefulness in practice. The first challenge is that TTP knowledge bases are optimized for recall, not precision; that is, TTP curators attempt to describe all procedures that have any possibility of being attack related, even if the same procedures are widely employed for innocuous purposes. An obvious example of this problem can be found in the "File Deletion" Technique [75] in MITRE ATT&CK – while file deletion may indicate the presence of evasive APT tactics, it is also a necessary part of benign user activities. As a result, EDR tools are prone to high volumes of false alarms [5, 6, 7, 8]. In fact, EDR tools are one of the key perpetrators of the "threat alert fatigue" problem that is currently plaguing the industry. A recent study found that the biggest challenge for 35% of security teams is keeping up with the sheer volume of alerts [76]. Consequently, the true attacks detected by EDR tools are at risk of being lost in the noise of false alerts.

The second challenge comes from the dubious nature of EDR-generated threat alerts. After receiving an alert, the first job of a security analyst is to determine the alert's veracity. For validation, security analysts review the context around the triggered alert by querying the EDR for system logs. Although EDR tools collect a variety of useful contextual information, such as running processes and network connections, the onus is on the security analyst to manually piece together the chain of system events. If the alert is deemed truly suspicious, the security analyst then attempts to recover and correlate various stages of the attack

through further review of enormous system logs. Security Indicator & Event Management (SIEM) products are often the interface through which this task is performed (e.g., Splunk [9]), allowing analysts to write long ad-hoc queries to join attack stages, provided that they have the experience and expertise to do so.

Long-term log retention is the third challenge for existing EDR tools. It is still commonplace for EDR tools to delete system logs soon after their capture. Logs are commonly stored in a small FIFO queue that buffers just a few days of audit data [77, 78], such that system events are commonly unavailable when investigating a long-lived attack. Even worse, unless an organization staffs a 24/7 security team, the audit data for an alert that fires over the weekend may be destroyed by Monday. This indicates that despite advancements in the efficiency of causal analysis, long-term retention of system log simply does not scale in large enterprises. Not only does this mean that EDR tools cannot reap the benefits of causal analysis during threat investigation, but it also means that current EDR tools lack the necessary context to understand the interdependencies between related threat alerts.

Based on data provenance, we introduce a new concept in this chapter which we call *Tactical Provenance* that can reason about the causal dependencies between EDR-generated threat alerts. Those causal dependencies are then encoded into a tactical provenance graph (TPG). The key benefit of TPG is that a TPG is more succinct than a classical whole-system provenance graph because it abstracts away the low-level system events for security analysts. Moreover, TPGs provide higher-level visualizations of multi-stage APT attacks to the analysts, which help to accelerate the investigation process.

To tackle the threat alert fatigue problem, we present methods of triaging threat alerts based on analysis of the associated TPGs. APT attacks usually conform to a "kill chain" where attackers perform sequential actions to achieve their goals [79, 80]. For instance, if the attacker wants to exfiltrate data, they must first establish a foothold on a host in the enterprise, locate the data of interest (i.e., reconnaissance), collect it, and finally transmit the data out of the enterprise. Our key idea is that *these sequential attack stages seen in APT campaigns can be leveraged to perform risk assessment.* We instantiate this idea in a threat score assignment algorithm that inspects the temporal and causal ordering of threat alerts within the TPG to identify sequences of APT attack actions. Afterward, we assign threat score to that TPG based on the identified sequences and use that threat score to triage TPGs.

To better utilize the limited space available on hosts for long-term log storage, we present a novel log reduction technique that, instead of storing all the system events present in the logs, maintains a minimally-sufficient skeleton graph. This skeleton graph retains just enough context (system events) to not only identify causal links between the existing alerts

but also any alerts that may be triggered in the future. Even though skeleton graphs reduce the fidelity of system logs, they still preserve all the information necessary to generate TPGs for threat score assignment, risk assessment, and high-level attack visualization.

We integrate our prototype system, RapSheet, into the Symantec EDR tool. We evaluated RapSheet with an enterprise dataset to show that RapSheet can rank truly malicious TPGs higher than false alarm TPGs. Moreover, our skeleton graph reduces the storage overhead of system logs by up to 87% during our experiments.

3.1 BACKGROUND & MOTIVATION

In this section, we first give background on the MITRE ATT&CK knowledge base and EDR tools. Then, we use a real-world attack scenario to illustrate the limitations of EDR tools and to motivate our work. Finally, we present our design goals and approach overview.

3.1.1 MITRE ATT&CK and EDR tools

MITRE ATT&CK is a publicly-available knowledge base of adversary tactics and techniques based on real-world observations of cyber attacks. Each tactic contains an array of techniques that have been observed in the wild by malware or threat actor groups. Tactics explain what an attacker is trying to accomplish, while techniques[1] and procedures[2] represent how an adversary achieves these tactical objectives (e.g., How are attackers escalating privileges? or How are adversaries exfiltrating data?) The MITRE ATT&CK Matrix [81] visually arranges all known tactics and techniques into an easy-to-understand format. Attack tactics are shown at the top of the matrix. Individual techniques are listed down each column. A completed attack sequence would be built by moving through the tactic columns from left (Initial Access) to right (Impact) and performing one or more techniques from those columns. Multiple techniques can be used for one tactic. For example, an attacker might try both an attachment (T1193) and a link (T1192) in a spearphishing exploit to achieve the Initial Access tactic. Also, some techniques are listed under multiple tactics since they can be used to achieve different goals.

One common use of MITRE ATT&CK tactics and techniques is in malicious behavior detection by Endpoint Detection and Response (EDR) tools. EDR tools serve four main

[1] Techniques are referenced in ATT&CK as `Txxxx` such as Spearphishing link is T1192 and Remote Access Tools is T1219. Description of these techniques is available at `https://attack.mitre.org/techniques/enterprise/`

[2] A procedure is a specific instantiation of a technique; in this paper we use the term "technique" to describe both techniques and procedures.

purposes in enterprises: 1) detection of potential security incidents, 2) scalable log ingestion and management, 3) investigation of security incidents, and 4) providing remediation guidance. To implement those capabilities, EDR tools record detailed, low-level events on each host including process launches and network connections. Typically, this data is stored locally on end hosts. Events that are of potential interest may be pushed to a central database for alerting and further analysis, during which additional events may be pulled from the endpoint to provide forensic context. EDR tools provide a rule matching system that processes the event stream and identifies events that should generate alerts. Major EDR vendors [71, 72, 73] already provide matching rules to detect MITRE ATT&CK TTPs; however, security analysts can also add new rules to detect additional TTPs at an enterprise where the EDR tool is deployed.

3.1.2 Motivating Example

We now consider a live attack exercise that was conducted by the Symantec's red team over a period of several days; this exercise was designed to replicate the tactics and techniques of the APT29 threat group. APT29 is one of the most sophisticated APT groups documented in the cyber security community [82]. Thought to be a Russian state-sponsored group, APT29 has conducted numerous campaigns with different tactics that distribute advanced, custom malware to targets located around the globe. Discovered attacks attributed to APT29 have been carefully analyzed by MITRE, yielding a known set of tactics and techniques that APT29 commonly use to achieve their goals [83]. In this exercise, different techniques were performed from that known set, ranging from Registry Run Keys (T1060) to Process Injection (T1055). These techniques allowed us to observe different MITRE tactics including persistence, privilege escalation, lateral movement, and defense evasion.

Limitations of EDR tools

Existing EDR tools excel at scalably identifying potentially malicious low-level behaviors in real-time. They can monitor hundreds or thousands of hosts for signs of compromise without event congestion. However, they suffer from some major usability and resource issues which we list below.

False-positive Prone. Existing EDR tools are known to generate many false alarms [5, 6, 7] which lead to the threat alert fatigue problem. The main reason for this high false alarm rate is that many MITRE ATT&CK behaviors are only sometimes malicious. For example, MITRE ATT&CK lists a technique called "File Deletion" T1107 under the "Defense

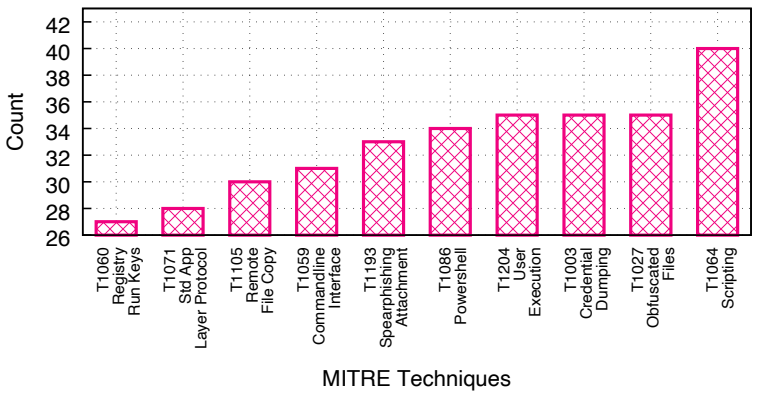


Figure 3.1: Top 10 techniques based on the number of times exploited by 93 MITRE-curated APT groups. 6 of these 10 techniques are benign in isolation and occur frequently during normal system execution.

Evasion" tactic. Finding this individual behavior and generating an alert is straightforward. But how would the analyst discern whether this file deletion is the result of normal system activity, or an attempt by an attacker to cover his tracks? Alerting on individual MITRE techniques generates false alarms and requires a human in the loop for alert validation.

To further quantify how many techniques from the MITRE ATT&CK knowledge-base can be benign in isolation, we took techniques used by 93 APT attack groups provided by MITRE and identified the most used techniques from these attack groups. Figure 3.1 shows the top ten most used techniques. After manual inspection, we found that 6 of 10 techniques may be benign in isolation, and in fact occur frequently during typical use. For example, the Powershell technique (T1086) can be triggered during a normal execution of applications like Chrome or Firefox. During our attacks simulation period, the Symantec EDR generated a total of 58,096 alerts on the 34 machines. We analyzed these alerts and found that only 1,104 were related to true attacks from the APT29 exercise and from other attack simulations we describe later. The remaining 56,992 were raised during benign activity, yielding a precision of only 1.9%.

Laborious Context Generation. To investigate and validate the triggered alerts, analyst usually write ad hoc queries using the SIEM or EDR tool's interface to generate context around alerts or to correlate them with previous alerts. Such context generation requires a lot of manual effort and time, which can delay investigation and recovery. Even after analysts have generated the context around an alert, it is difficult to understand the progression of the attack campaign by looking at system-level events. Depicting these events in a graph helps to show the causal relationships, but the volume of information is still overwhelming. Note that certain EDR tools, such as CrowdStrike Falcon [84] provide interfaces to only

Figure 3.2: Part of the APT29 attack provenance graph. We zoomed-in on two threat alerts from this attack, and excluded the network connections and registry operations from this graph for presentation purposes. In the complete graph, there are total 2,342 edges and 1,541 vertices. In this graph, and the rest of the paper, we use boxes to represent processes (count=79), diamonds to represent sockets (count=750), and oval nodes to represent files (count=54), registries (count=132), kernel objects (count=30), and modules (count=496). Edges represent casual relationships between the entity nodes, and red edges represent threat alerts (count=26).

get the chain of process events that led to the triggered alert. These process chains do *not* capture information flow through system objects (e.g., files, registries). As a result, such EDR tools can not aggregate causally related alerts that are associated with system objects, leading to incomplete contexts.

During our exercise, APT29 generated 2,342 system events such as process launches and file creation events. Figure 3.2 shows a classical whole-system provenance graph for all the events related to APT29. The unwieldy tangle of nodes and edges in the figure demonstrates how daunting it can be for a security analyst to explore and validate a potential attack and understand the relationship between alerts.

Storage Inefficiency. EDR tools constantly produce and collect system logs on the end hosts. These system logs can quickly become enormous [20, 22]. In our evaluation dataset, the EDR recorded 400K events per machine per day from total of 34 end hosts, resulting in 35GB worth of system logs with a total of 40M system events. Note that the database used to store the events on hosts performs light compression, resulting in on-disk sizes roughly half this size. Retaining those system logs can become costly and technically challenging over longer periods. Further, for enterprises, it is important to clarify how long logs will be stored for and plan for the resulting financial and operational impact. For example, keeping log data for a week may be inexpensive, but if an attack campaign spans more than a week (which is common [1, 2, 85]), then the company will lose critical log data necessary for forensic investigation.

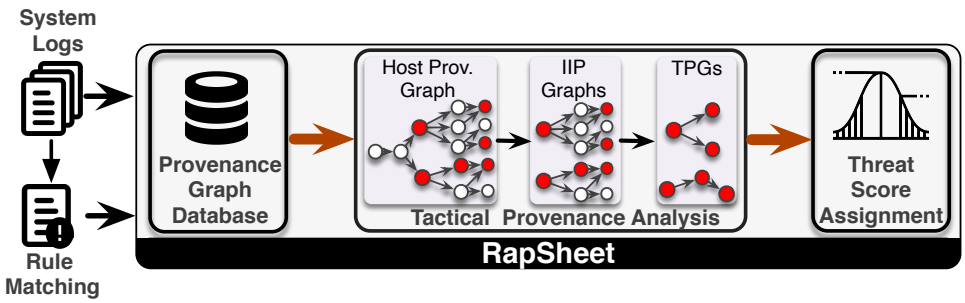


Figure 3.3: Overview of RapSheet architecture (Section 3.1.3)

We surveyed the white papers and manuals of the top 5 EDR tools curated by Gartner [74]. In these white papers, we specifically looked for techniques used by these EDR tools for log retention. We found that *no* EDR tool currently describes any meaningful log retention techniques that can best utilize the limited storage for the investigation of long-lived APTs. Instead, those EDR tools use a FIFO queue that depending on the EDR vendor's retention policies buffers only a few days of system logs. For example, by default, Symantec's EDR allocates 1GB of space on each host which is sufficient for a couple of days or perhaps a week's worth of logs. The oldest logs are purged when this limit is reached. Events that are pushed to the server are also purged, with the oldest 10% of events deleted when used storage capacity reaches 85% [77].

3.1.3 Our Approach

A high-level overview of our system, RapSheet, is shown in Figure 3.3. Full details will be given in the next section, but we overview the approach here. First, RapSheet performs rule matching on system logs to identify the events that match MITRE ATT&CK behaviors. In our APT29 exercise, we were able to match techniques T1060, T1086, T1085, T1055, T1082, T1078, T1076, T1040 against logs. Each rule match signifies an alert of a possible threat behavior. Next, we generate a provenance graph database from the logs. During the graph generation, we annotate the edges (events) that match the MITRE ATT&CK techniques in the previous step. Figure 3.2 shows the provenance graph for the APT29 engagement.

Once the construction of the provenance graph with alert annotations is done, we generate a tactical provenance graph (TPG) which is a graph derived from the provenance graph that shows how causally related alerts are sequenced. To generate a TPG, we first identify the *initial infection point* (IIP) vertex, i.e., the first vertex in the timeline that generated a threat alert. Then we find all the alerts in the progeny of the IIP vertex using forward tracing. Finally, extraneous system events are removed from this progeny graph, forming what we

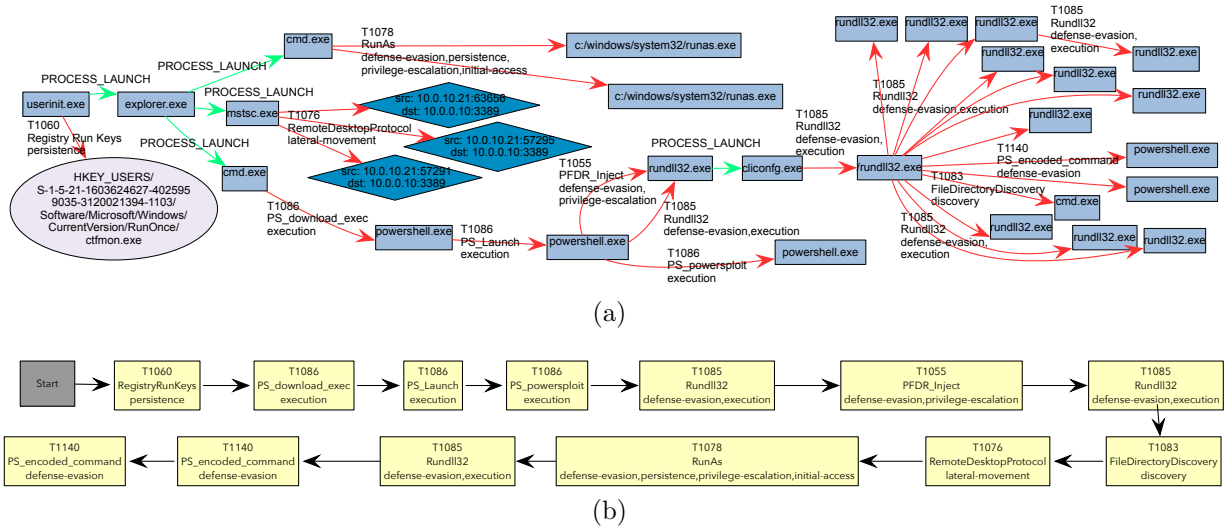


Figure 3.4: APT29 attack scenario. **(a)** IIP Vertex graph generated by RapSheet. Threat alert edges are annotated with the MITRE technique ID, technique name, and tactic name. "PS" stands for PowerShell. **(b)** Tactical Provenance Graph (TPG) for APT29 attack after applying readability pass. RapSheet generated TPG is 2 orders of magnitude smaller than the classical provenance graph shown in Figure 3.2

call the *IIP graph*. Figure 3.4a shows the IIP graph for the APT29 attack. After that, we perform threat score assignment.

The key idea behind our threat score assignment algorithm is to *use temporal ordering between all the causally related alerts (i.e., all the alerts in the IIP graph) to rank the alerts that conform to the MITRE ATT&CK kill chain higher than the alerts that appear in an arbitrary order*. However, ordering information for alerts on different paths is not immediately apparent in the IIP graph. To remedy this, we perform a happens-before analysis to find temporal orderings between the different alerts present in the IIP graph which gives us a TPG. Figure 3.4b shows the TPG for the APT29 attack scenario. After that our threat score assignment algorithm finds ordered subsequences of alerts from the TPG that conform to the MITRE kill chain and uses these to assign a severity score for alert prioritization.

3.2 TACTICAL PROVENANCE ANALYSIS & ALERT TRIAGE

3.2.1 Log Collection

EDR tools collect system logs on each host in the enterprise. For Linux hosts, our underlying EDR uses Linux Audit framework [56] while for Windows it uses ETW [55] as well as custom system call hooking. This is standard for most EDR tools [86]. System logs contain

low-level system events including process launches and file operations. Those system events capture causal relationships between different system entities. For example, in Linux the causal relationship between a parent process creating a child process is represented by an event generated by capturing calls to `sys_clone()`. Once those system logs are collected on each host they are processed into a JSON format.

We note that we supplemented the events collected by our underlying EDR with logs of Asynchronous Local Procedure Call (ALPC) messages which we collected separately on Windows hosts. ALPC is the mechanism that Windows components use for inter-process communication (IPC) [87]. After running real-world attack scenarios on Windows machines, we realized that many of the attacks manifest in part through system activities that are initiated using ALPC messages. Missing those causal links can undermine the forensic investigation, as the provenance graph becomes disconnected without them. Note that previous papers [11, 14, 30, 88, 89] on Windows provenance do not capture ALPC messages, resulting in disconnected provenance chains.

3.2.2 Rule Matching

Generating alerts for individual MITRE techniques is a feature of most EDR tools, including the one that we use in our experiments. Because of RapSheet's novel use of TPGs for grouping, scoring, and triaging alerts, we are able to include even the most false-positive-prone MITRE techniques as alerts without overwhelming an analyst. In our experiments, we use a default set of MITRE rules that was provided by the Symantec EDR tool, and we supplemented these with additional rules for MITRE techniques that were not already covered. Users can easily extend our system by adding new rules for additional TTPs. Moreover, our rule matching only relies on events that are commonly collected by EDR tools or readily available from commodity auditing frameworks.

Listing 3.1: Example MITRE technique matching rule.

```
IF EXISTS E WHERE E.tgtType = 'network' AND
E.action = 'connect' AND E.dstPort = 3389
THEN ALERT(E.actorProc, 'T1076')
```

As is described next, the low-level system events will form edges in a provenance graph. In RapSheet, we annotate the edges that triggered an alert with the alert information (e.g., the MITRE technique ID). Some rules provided by the EDR vendor generate alerts for behaviors not covered by the MITRE ATT&CK, which we ignore these for the purposes of this work. For our example attack scenario described in Section 3.1, the threat alert annotated as **Alert**

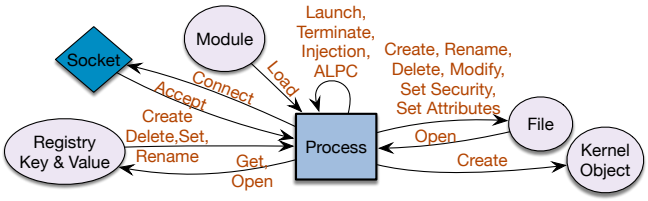


Figure 3.5: Data model of our provenance graph database. Vertices represent the system entities (actors and objects) while the edges represent the causal dependency. Edges are annotated with the timestamp of event occurrence and event type.

B in Figure 3.2 matched the rule (syntax simplified for clarity) shown in Listing 3.1.

3.2.3 Provenance Graph Database

The system logs on each host are parsed into a graph structure called a provenance graph. The provenance graph generated by RapSheet is similar to previous work on provenance graphs [10, 11, 12, 16, 33] with some new additions to reason about MITRE ATT&CK tactics. Our provenance graph data model is shown in Figure 3.5. We have two types of vertices: process vertex type and object vertex type which includes files, registry, etc. The edges that connect these vertices are labeled with an event type that describes the relationship between the connected entities and the timestamp of event occurrence. Moreover, process vertices are marked with start and terminate time which allows us to check if a process is still alive during our analysis.

We also implemented a summarization technique from previous work, *causality-preserved reduction* [20, 22] in our provenance graph database. This technique merges the edges between two vertices that have the same operation and keeps only one edge with the latest timestamp. For example, most operating systems and many EDRs produce several system-level events for a single file operation. RapSheet aggregates those events into a single edge in the provenance graph. This technique has been shown to reduce the size of the provenance graph while still preserving the correctness of causal analysis.

3.2.4 Tactical Provenance Analysis

Given a list of triggered alerts and host provenance graphs, we find all the *initial infection point* (IIP) vertices in the graphs. We define an IIP to be a vertex that meets two conditions: (i) it corresponds to a process that generated an alert event e_a, and (ii) a backward trace from e_a in the provenance graph contains no other alert events. Note that there can be multiple IIP vertices in a given provenance graph. Intuitively, we are finding the earliest

point that potentially suspicious behavior occurred on a given provenance chain. The IIP represents the process that exhibited this behavior. If it turns out that e_a was the first step in a multistage attack, then the remainder of the attack will be captured by future alerts generated by this process and its progeny. This gives us an effective way to group correlated alerts. For each IIP vertex, we generate a graph that is rooted at the IIP. We call this an IIP graph and define it as follows:

Definition 3.1. IIP Graph Given a provenance graph $G < V, E >$ and alert event e_a incident on IIP Vertex v_a, the IIP Graph $G' < V', E' >$ is a graph rooted at V_a where $e \in E'$ iff e is causally dependent on e_a and e is either an alert event or an event that leads to an alert event.

We generate the IIP graph by issuing a forward tracing query from the IIP vertex, producing a progeny provenance graph containing only events which happened after that first alert event incident on the IIP vertex. We then perform a pruning step on this subgraph, removing all provenance paths originating from the IIP that do not traverse an alert edge. Each path in the resulting, pruned graph contains at least one alert event. In Algorithm 3.1, Lines 1-13 show the IIP graph generation process. For our attack scenario example from Section 3.1, the pruned progeny graph rooted at the IIP is shown in Figure 3.4a.

This IIP graph based approach is a key differentiating factor that sets RapSheet apart from the path-based approach to alert triage in NoDoze [30] and the full graph approach in Holmes [89]. A path-based approach fails to correlate alerts that are causally related but appear on different ancestry paths. For example, after initial compromise, an attacker can launch several child processes, with each child generating its own, separate path. Even though all child paths are causally related, the path-based approach will fail to correlate alerts on the separate paths. On the other hand, Holmes' full graph approach requires a normal behavior database and other heuristics to reduce false alarms from benign activities before threat score assignment. RapSheet does not require a normal behavior database, rather we rely on extracting certain subgraphs (the IIP graphs) and assigning scores based on well-known attacker behaviors, which alleviates the problem of false alarms (further discussed in Section 3.3).

The IIP graph captures the temporal ordering between events on the same path. However, when reasoning about the overall attack campaign, we are not concerned with, e.g., which attacker-controlled process takes a given action. Instead, we want to capture the temporal order of all alerts contained in the IIP graph, which better reflects attacker intent. Because this graph may consist of multiple paths, we need a way to capture ordering between edges on different paths. To achieve this goal, we transform the IIP graph into a new graph in

which each vertex is an alert event and edges indicate the temporal ordering between alerts based on a happens-before relationship [90]. We call these edges sequence edges, and they are defined as follows:

Definition 3.2. Sequence Edge. A sequence edge (e_a, e_b) exists between two alerts e_a and e_b iff any of the following hold:
(a) e_a and e_b are alerts on the same host and on the same provenance path and e_a causally preceded e_b; or
(b) e_a and e_b are alerts on the same host and the vertex timestamp of e_a is less than the vertex timestamp of e_b or
(c) e_a had an outgoing `Connect` event edge on one host, while e_b has the corresponding `Accept` edge on the receiving host.

In other words, for events that happen on the same machine, we can use the event timestamps to generate sequence edges. For events on different machines, we can use communication between the machines to generate the happens-before relationship (events before a packet was sent on one machine definitely happened before events that happened after the packet was received on the other machine). In the end, we generate a graph (Algorithm 3.1 Lines 14-24) which we call a tactical provenance graph whose formal definition is as follows:

Definition 3.3. Tactical Provenance Graph. A tactical provenance graph TPG can be defined as a pair (V, E), where V is a set of threat alert events and E is a set of sequence edges between the vertices.

As defined above, the TPG is already useful for analysts to visualize multi-stage APT campaigns because it shows temporally ordered and causally related stages of an attack without getting bogged down in low-level system events. However, the tactical provenance graph may not be as succinct as the analyst would like, since MITRE techniques may be matched repeatedly on similar events, such as a process writing to multiple sensitive files or a process sending network messages to multiple malicious IP addresses. This can add redundant alert event vertices in the tactical provenance graph. To declutter the TPG, we perform a post-processing step where we aggregate the alert vertices ascribing the same technique if they were triggered by the same process. Note that for events on a single host, without cross-machine links, the TPG is a single chain. An illustration of this post-processing step is given in Figure 3.4a. While the IIP shows `mstsc.exe` triggering three lateral movement alerts, the TPG in Figure 3.4b only has one lateral movement vertex.

Algorithm 3.1: Tactical Provenance Analysis

Inputs : Raw provenance graph $G(V, E)$; Alert Events AE
Output: List of Tactical Provenance Graphs $List_{TPG}$

$AE' \leftarrow \{ae : time(ae)\}, ae \in AE$, sort by timestamp in asc. order
$Seen \leftarrow \emptyset$, set of seen alert events
$List_{IIP} \leftarrow \emptyset$, List of IIP Vertex Graphs
for $ae : AE', ae \notin Seen$ **do**
    $Seen \leftarrow Seen \cup \{ae\}$ /* return all forward tracing paths from input event using DFS */
    $Paths \leftarrow$ ForwardPaths(ae)
    $IIPG \leftarrow \emptyset$, IIP graph
    for $path : Paths$ **do**
        /* return all alert events in the input provenance path */
        $alerts \leftarrow$ GetAlertEvents($path$)
        /* keep only those paths in IIP graph with at least one alert */
        if $alerts \neq \emptyset$ **then**
            $IIPG \leftarrow IIPG \cup path$
            $Seen \leftarrow Seen \cup alerts$
    $List_{IIP} \leftarrow List_{IIP} \cup IIPG$
$List_{TPG} \leftarrow \emptyset$, List of TPGs to return
for $IIPG : List_{IIP}$ **do**
    $TPG \leftarrow \emptyset$, tactical provenance graph
    $alerts \leftarrow$ GetAlertEvents($IIPG$)
    /* sort alerts according to Happens Before rules */
    $alerts_{hb} \leftarrow \{a : time(a)\}, a \in alerts$
    /* Loop over sorted alerts, two at a time */
    for $ae_1, ae_2 : alerts_{hb}$ **do**
        $V \leftarrow ae_1$
        $V' \leftarrow ae_2$
        $TPG \leftarrow TPG \cup (V, V')$ /* add sequence edge */
    /* Post process the TPG for readability */
    $TPG \leftarrow$ ReadabilityPass(TPG)
    $List_{TPG} \leftarrow List_{TPG} \cup TPG$
return $List_{TPG}$

3.3 THREAT SCORE ASSIGNMENT

A key goal of RapSheet is to group alerts and assign them a threat score that can be used to triage those contextualized alerts. Because some alerts are more suspicious than others, we pursued a scoring mechanism that incorporated a risk score of the individual alerts. Where available, we used information published by MITRE to assign those scores to individual alerts.

Many of the MITRE ATT&CK technique descriptions include a metadata reference to a pattern in the Common Attack Pattern Enumeration and Classification (CAPEC) [91] knowledge base. The CAPEC pattern entries sometimes include two metrics for risk assessment: "Likelihood of Attack" and "Typical Severity". Each of these is rated on a five category scale of `Very Low, Low, Medium, High, Very High`. The first metric captures how

likely a particular attack pattern is to be successful, taking into account factors such as the attack prerequisites, the required attacker resources, and the effectiveness of countermeasures that are likely to be implemented. The second metric aims to capture how severe the consequences of a successful implementation of the attack would be. This information is available on MITRE's website, as well as in a repository of JSON files [92] from which we programmatically extracted the scores.

For some MITRE techniques, no CAPEC reference is provided, or the provided CAPEC reference has no likelihood and severity scores. In these cases, we fall back on a separate severity score that was provided by the EDR vendor, normalized to our fifteen point scale. We converted the descriptive values for each metric into a numeric scale of one to five, and combined the two metrics together. We give the severity score a higher weight than the likelihood score since we are defending against advanced adversaries that have many resources at their disposal to effectively execute techniques that might be considered unlikely due to their difficulty or cost. The resulting threat score for each individual alert is:

$$TS(\text{technique}) = (2 * SeverityScore) + LikelihoodScore \tag{3.1}$$

For example, the MITRE technique called *Registry Run Keys / Startup Folder* (T1060) [93] refers to the attack pattern called *Modification of Registry Run Keys* (CAPEC-270) [94] which assigns a likelihood of attack of "medium" and a severity of "medium". Thus, we assign an alert that detects technique T1060 a score of nine out of a possible fifteen ($TS(\text{T1060}) = 2 * 3 + 3 = 9$).

Next, we explain different schemes that we used to combine individual alert scores into an overall threat score.

3.3.1 Limitations of Path-Based Scoring Schemes

To aggregate scores, we first tried an approach based on grouping and scoring alerts using a single, non-branching provenance path as was proposed by Hassan et al. in [30]. For each alert, we generated the backward tracing path and then aggregated the scores that occurred on that path. We tried different aggregation schemes such as adding the individual alert scores or multiplying them, with and without technique or tactic deduplication. Unfortunately, we realized during our experiments that the path-based approach was not capturing the entire context of the attacks in some situations. This led us to explore another approach to grouping and scoring alerts.

3.3.2 Graph-Based Scoring Schemes

To capture the broader context of a candidate alert, we generate the TPG for the candidate alert which is derived from the subgraph rooted at the shallowest alert in the candidate's backward tracing provenance path as described in Section 3.2.

The key insight behind our proposed scheme is that we would like to maximize the threat score for TPGs where the alerts are consistent with an attacker proceeding through the ordered phases of the tactical kill chain defined by MITRE. We formalize this intuition in a scoring algorithm as follows. The sequence edges in the TPG form a temporally ordered sequence of the graph's constituent alerts. We find the longest (not necessarily consecutive) subsequence of these ordered alerts that is consistent with the phase order of MITRE's tactical kill chain. We then multiply the scores of the individual alerts in this subsequence to give an overall score to the TPG. If there are multiple longest subsequences, we choose the one that yields the highest overall score. More formally:

$$TS(\text{TPG}) = \max_{\mathbf{T^i} \in \mathfrak{T}} \prod_{T_j^i \in \mathbf{T^i}} TS(T_j^i) \tag{3.2}$$

In Equation 3.2, $\mathfrak{T}$ is the set of all longest subsequences in TPG consistent with both temporal and kill-chain phase ordering. Note that an attacker cannot evade detection by introducing out-of-order actions from earlier, already completed stages of the attack. RapSheet's scoring approach will simply ignore these actions as noise when finding the longest subsequence of alerts from the TPG, which need not be consecutive.

3.4 LOG REDUCTION

System logs enable two key capabilities of EDR tools: 1) threat alert triage based on alert correlation and 2) after-the-fact attack investigation using attack campaign visualization. Thus, EDR tools need to retain these logs long enough to provide these capabilities. However, system logs can become enormous quickly in large enterprises, making long-term retention practically prohibitive. As mentioned in Section 3.1, most EDR tools store logs in a limited FIFO buffer, destroying old logs to make space for new logs. Unfortunately, this naive log retention strategy can lose critical information from older logs. So, it is important to use this limited memory efficiently.

We propose a novel technique to reduce the fidelity of logs while still providing the two key EDR capabilities. To provide these key capabilities, we need to ensure that we can generate the TPG from the pruned graph. Once we have the TPG, we can derive correlations between

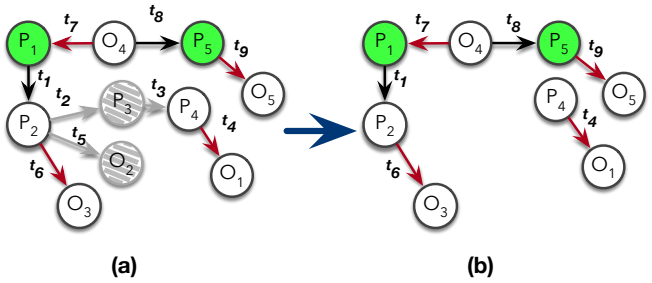


Figure 3.6: Graph reduction example. After every configurable time interval, RapSheet runs graph reduction and store only skeleton graph which preserves the linkability between current and future tactics.

alerts, assign threat scores to correlated alerts and provide high-level visual summaries of attacks to the security analyst.

For our graph reduction algorithm, we assume the properties of the provenance graph and backward tracing graph described in Section 3.2.3. We also assume all the alert events in the provenance graph are incident to at least one process vertex. Based on these properties, we propose the following two rules to prune the provenance graph at any point in time while preserving TPG-based alert correlation.

Rule#1: Remove object vertex O iff there are no alert events in the backward tracing graph of O and there are no alert event edges directly connected to O.

This rule ensures that O is not currently part of any IIP graph derived from the current provenance graph. If it were, then it either would be directly involved in an alert (i.e., there would be an alert edge incident to O), or it would be on a path from some IIP vertex to some alert edge, which entails that the alert incident to that IIP vertex would be in O's backward tracing graph. Note that even if there is a live process vertex in the ancestry of object O, and that process generates an alert event E_1 in the future, this new alert event will have a timestamp later than the edges currently leading to O. Hence, O would not be part of the IIP graph containing E_1.

To explain our graph reduction algorithm we use an example provenance graph shown in Figure 3.6**(a)**. Vertices labeled with a P represent processes while those with an O represent object vertices. The red edges indicate alerts, green vertices show live processes at the time of reduction, and edges are marked with ordered timestamps t_1 to t_9. Gray vertices and edges show candidates for removal according to Rule#1 and Rule#2.

The only candidate for object vertex reduction is O_2 since it satisfies all the conditions of Rule#1. The backward tracing graph of O_2 consists of vertices $\{P_2, P_1\}$ and the edges with timestamps $\{t_5, t_1\}$, which do not have any alert events. Thus, we can safely remove O_2 and

the edge with timestamp t_5 from the graph without losing any connectivity information for current or future alerts. Note that the edge with timestamp t_7 will not be included in the backward tracing graph because it happened after t_5. After graph reduction, if some process vertex reads or writes to the object O_2, then vertex O_2 will reappear in the provenance graph. Next, we discuss how to prune process vertices from the graph.

Rule#2: Remove process vertex P iff: i) there are no alert events in the backward tracing graph of P, ii) there are no alert event edges directly connected to P and iii) process P is terminated.

The first two conditions of Rule#2 have the same reasoning as Rule#1. In addition, we have to ensure that process P is terminated so that it does not generate new alerts which will become part of an IIP graph. In the example shown in Figure 3.6**(a)**, process P_3 is terminated, has no alert event in its backward tracing graph, and does not have any incident edges that are alert events. Thus, we can safely remove the process vertex P_3 from the graph along with the edges that have timestamp $\{t_2, t_3\}$.

By applying these two reduction rules to a given provenance graph, RapSheet generates a space-efficient skeleton graph which can still identify all the causal dependencies between alerts and can generate exactly the same set of TPGs (procedure described in Section 3.2.4) as from the classical provenance graph. Figure 3.6**(b)** shows the skeleton graph for our example graph.

Properties. A skeleton graph generated by RapSheet will not have any false positives, that is, TPGs generated from the skeleton graph will not have alert correlations that were not present in the original provenance graph. This is clear since RapSheet does not add any new edges or vertices during the reduction process. Furthermore, a skeleton graph generated by RapSheet will not have any false negatives, meaning it will capture all alert correlations that were present in the original provenance graph. This follows from the properties of provenance and our backward tracing graphs. The reduction rules ensure that, at the time of reduction, the removed nodes and edges are not part of any IIP graph. And since our backward traces include only events that happened before a given event, they would not be part of any future IIP graph.

Retention Policy. To provide log reduction and prevent storage requirements from growing indefinitely, enterprises can run the graph reduction algorithm at a configurable retention time interval. This configuration value must be long enough for alert rule matching to complete. The retention policy can be easily refined or replaced according to enterprise needs. The configured retention interval controls how long we store high-fidelity log data (i.e., the unpruned graph). RapSheet's backward tracing and forward tracing works seamlessly

over the combined current high-fidelity graph and the skeleton graph that remains from prior pruning intervals.

3.5 EVALUATION

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

RQ1 How effective is RapSheet as an alert triage system?

RQ2 How fast can RapSheet generate TPGs and assign threat scores to TPGs?

RQ3 How much log reduction is possible when using skeleton graphs?

RQ4 How well does RapSheet perform against realistic attack campaigns?

3.5.1 Implementation

We used Apache Tinkerpop [95] graph computing framework for our provenance graph database. Tinkerpop is an in-memory transactional graph database and provides robust graph traversal capabilities. We implemented the three RapSheet components (tactical graph generation, threat score assignment, and graph reduction) in 6K lines of Java code. We use a single thread for all our analyses. We generate our provenance graphs in GraphViz (dot) format which can be easily visualized in any browser. Our implementation interfaces with Symantec EDR. Symantec EDR is capable of collecting system logs, matching events against attack behaviors, and generating threat alerts.

3.5.2 Experiment Setup & Dataset

We collected system logs and threat alerts from 34 hosts running within Symantec. The logs and alerts were generated by Symantec EDR which was configured with 67 alert generating rules that encode techniques from the MITRE ATT&CK knowledge-base. In our experiments, we turned off other EDR rules that did not relate to MITRE ATT&CK. During all experiments, RapSheet was run on a server with an 8-core AMD EPYC 7571 processor and 64 GB memory running Ubuntu 18.04.2 LTS.

Our data was collected over the period of one week from hosts that were regularly used by members of a product development team. Tasks performed on those hosts included web

browsing, software coding and compilation, quality assurance testing, and other routine business tasks. Due to variations in usage, some machines were used for only one day while others logged events every day during data collection week. In total, 35GB worth of (lightly compressed) logs with around 40M system events were collected. On average, each host produced 400K events per machine per day. We describe further characteristics of our dataset in the next subsection 3.5.3.

During the experimental period, we injected attack behaviors into three different hosts. The attack behaviors correspond to three different attack campaigns, two based on real-world APT threat groups (APT3 and APT29) and one custom-built data theft attack. These simulated attacks were crafted by an expert security red-team. The underlying EDR generated 58,096 alerts during the experiment period. We manually examined the alerts from the machines which were targeted by the simulated attacks to determine that 1,104 alerts were related to simulated attacker activity. The remaining alerts were not associated with any of the simulated attacks and we consider them to be false positives.

3.5.3 Dataset Characterization

In this section, we characterize dataset that we used in our evaluation. We collected 40M system monitoring event from 34 hosts in a real-world enterprise environment. These host machines were used by employees daily for web browsing, software coding and compilation, quality assurance testing, project management, and other routine business tasks. We used 67 total alert rules to detect various MITRE ATT&CK techniques in our experiments. Of these rules, some were written by us, while the other were included by default in the Symantec EDR software.

First, we look at how often the various MITRE ATT&CK technique and tactic rules caused alerts on the hosts in our experiment. Figure 3.7 shows which MITRE ATT&CK techniques were matched, how many times, and what proportion of the alerts for each technique were related to a true attack. We can see from the figure that rules for techniques like RunDLL32 (T1085) and Scripting (T1064) generated many alerts, but have very low true positive rates since these techniques are commonly used for benign purposes. On the other hand, techniques like "Change File Association" (T1042) and "System Service Discovery" (T1007) were triggered many times and have high true positive rate because these techniques usually only happen during malicious activity. Thus, these techniques can be strong indication of an attack campaign.

Figure 3.8 shows the tactics to which the alerting techniques in our data belong. During evaluation, we observed 10 out of the 12 tactics defined by MITRE ATT&CK. As is evident

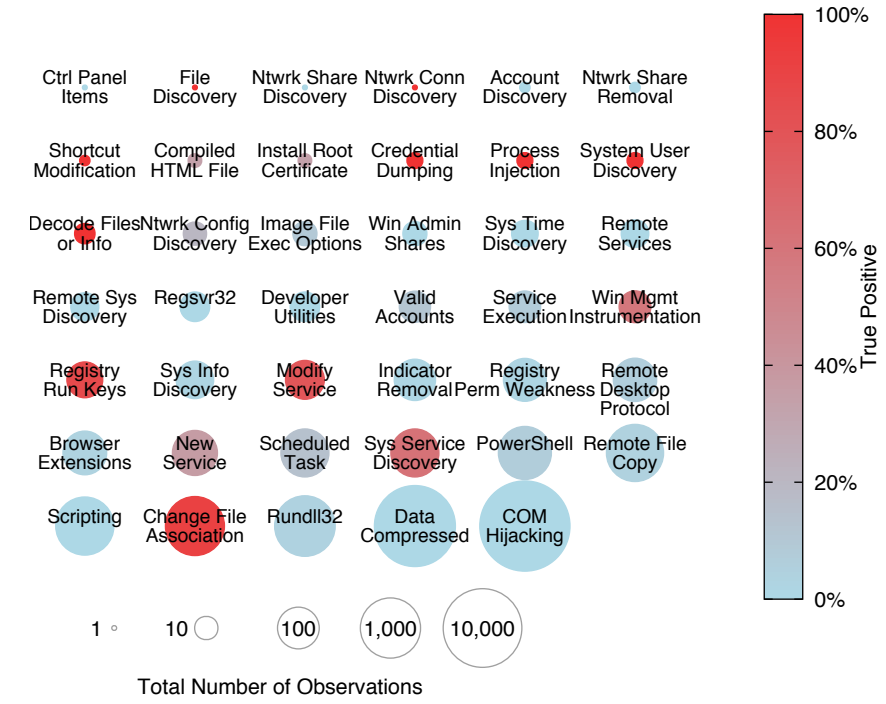


Figure 3.7: Number of matched MITRE ATT&CK techniques during our evaluation with their true positive rates.

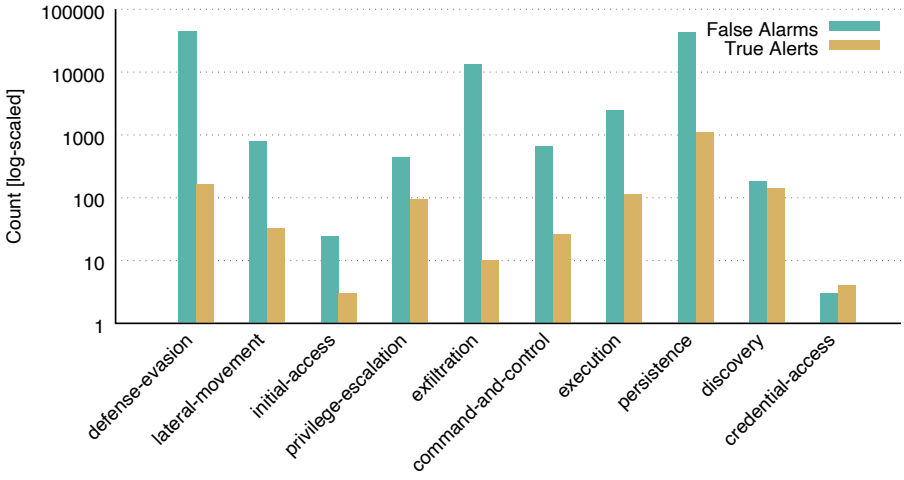


Figure 3.8: Number of matched MITRE ATT&CK tactics during our evaluation.

from the graph, there are certain tactics, such as "Exfiltration" and "Defense Evasion", are more false-positive prone. Others, such as "Discovery", still have many false alarms, but have a more balanced distribution.

Figure 3.9 shows the number of vertices and edges in provenance graph database for each of 34 hosts in our evaluation. We see that all hosts have a similar number of edges and vertices except for two hosts. From these two hosts, we had only few hours worth of logs.

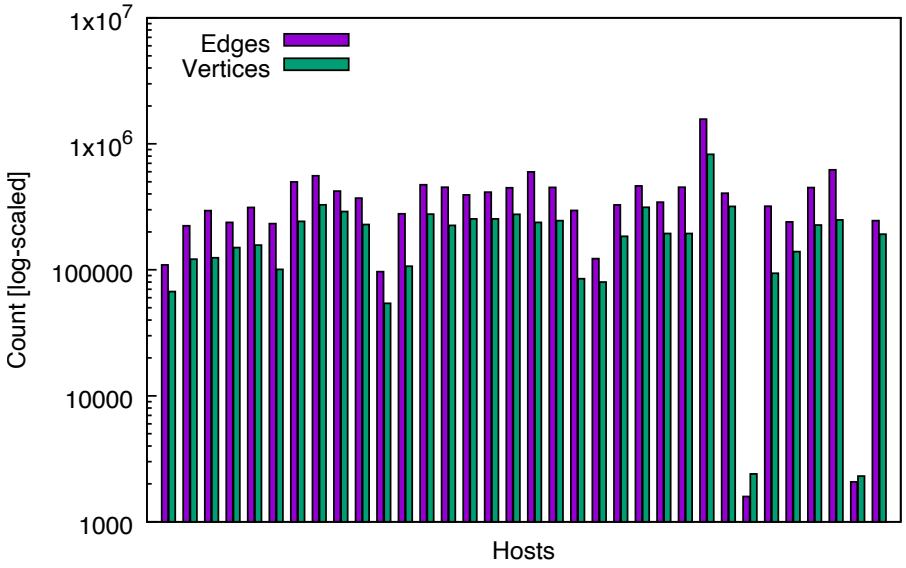


Figure 3.9: Number of vertices and edges in the provenance graph for each of 34 hosts in our evaluation.

3.5.4 Effectiveness

The first research question of our evaluation is how effective RapSheet is as an alert triage tool. In our experiment, we used the EDR tool to monitor hosts for MITRE ATT&CK behaviors and generate alerts. We then manually labeled these alerts as true positives and false positives based on whether the log events that generated the alert were related to simulated attacker activity. This labeled set is used as the ground truth in our evaluation. Then, we used RapSheet to automatically correlate these alerts, generate TPGs, and assign threat scores to TPGs.

Of the 1,104 true alerts and 56,992 false alarms generated during our experiments, RapSheet correlated these alerts into 681 TPGs. Of these, 5 were comprised of true alerts and 676 contained only false alarms.[3] We then calculated threat scores for these TPGs and sorted them according to their score. We tried two different scoring schemes. For the first scheme, we assigned scores to each TPG using a strawman approach of multiplying the threat scores of all alerts present in the TPG. However, since TPGs may contain duplicate alerts, we normalize the score by combining alerts which have the same MITRE technique, process, and object vertex. For the second scheme, we used the scoring methodology described in Section 3.3.

Different true positive rates (TPRs) and false positive rates (FPRs) for the scoring schemes

[3]Three out of five truly malicious TPGs were related to the APT29 simulation, which the red team performed three times during the week with slight variations. The other two attack campaigns resulted in one TPG each.

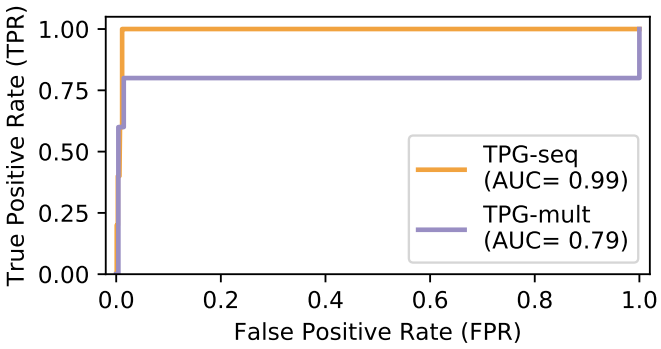


Figure 3.10: ROC curve for our experiments. We tried two different schemes to rank TPGs. TPG-Seq means sequence-based scoring while TPG-mult means strawman approach of score multiplication.

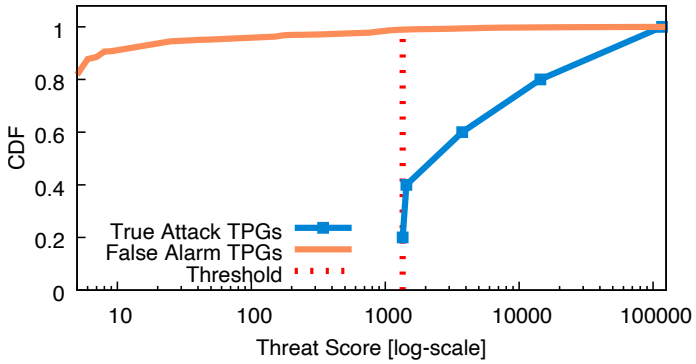


Figure 3.11: CDF of threat scores for false alarm and true attack TPGs.

above are shown in the ROC graph in Figure 3.10. Our sequence-based scoring scheme was more effective than the other scheme. Figure 3.11 shows the cumulative distribution function for ranked true attack and false alarm TPGs based on threat scores. When we set a threshold (shown with a vertical red line) that captures 100% of true positives, we can remove 97.8% of false TPGs since all true attack TPGs are scored significantly higher than most false alert TPGs. At this threshold, RapSheet has a 2.2% FPR. Note that the goal of RapSheet is not to eliminate false TPGs from consideration, but to prioritize TPG investigation based on their threat score. The threshold is a configurable parameter and can be set more conservatively or aggressively based on the goals of a particular enterprise security team.

Table 3.1 summarizes the ranking of top 16 threat scoring TPGs out of total 681 TPGs in our evaluation. This list contains all the 5 truly malicious TPGs that are present in our evaluation. In this table, the first column represents the root vertex ID given by RapSheet. Recall that TPG is identified by the IIP root vertex. The second column shows where the TPG was a false alarm or truly malicious. The third column shows the threat score given by

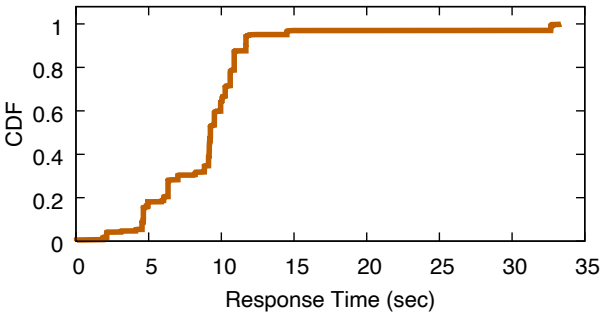
Table 3.1: Top 16 threat scoring TPGs out of total 681 TPGs.

TPG ID	Category	Threat Scores	Threat Alerts	Longest Ordered Subsequence of Tactics
052c89	False Alarm	125000	54	execution, persistence, privilege-escalation, defense-evasion, discovery, lateral-movement, command-and-control
e431ac	True Attack	116640	992	execution, persistence, privilege-escalation, defense-evasion, credential-access, discovery
69f88c	False Alarm	25000	30	execution, persistence, privilege-escalation, defense-evasion, discovery, lateral-movement
2e91b2	False Alarm	15625	52403	execution, persistence, privilege-escalation, defense-evasion, lateral-movement, exfiltration
c17d94	True Attack	14400	26	persistence, privilege-escalation, defense-evasion, discovery, lateral-movement
9b1f4a	False Alarm	7350	12	execution, persistence, privilege-escalation, defense-evasion, discovery
3f3fa5	False Alarm	5250	26	persistence, privilege-escalation, defense-evasion, discovery, exfiltration
08f25f	False Alarm	4375	45	execution, persistence, privilege-escalation, lateral-movement, exfiltration
ba6b01	False Alarm	3750	11	execution, persistence, privilege-escalation, defense-evasion, discovery
b464e4	True Attack	3750	77	persistence, privilege-escalation, defense-evasion, discovery, exfiltration
8d88e3	False Alarm	3125	127	execution, persistence, defense-evasion, discovery, exfiltration
d68b64	False Alarm	3125	16	initial-access, execution, persistence, defense-evasion, exfiltration
3fb85e	False Alarm	1600	44	execution, persistence, lateral-movement, exfiltration
ae5f39	False Alarm	1600	64	execution, persistence, lateral-movement, command-and-control
e448f1	True Attack	1440	13	execution, defense-evasion, discovery, lateral-movement
0c1d5e	True Attack	1350	48	execution, defense-evasion, discovery, lateral-movement

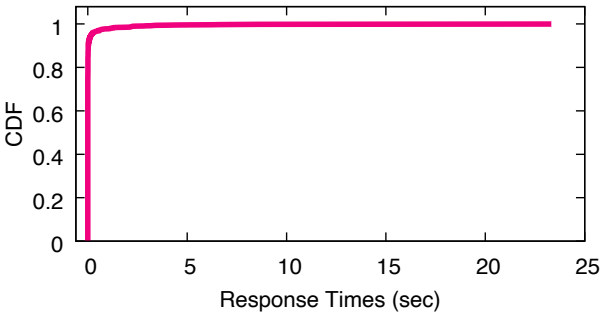
RapSheet to the TPG. The fourth column shows the number of threat alerts present in the corresponding TPG. Finally, the fifth column represents the longest ordered sub-sequence extracted by RapSheet from the TPG that gave the highest threat score.

3.5.5 Response Times

To answer RQ2, we measured the TPG generation query response (turn-around) time for all the alerts in our evaluation dataset. We divided the response time of TPG generation queries into two parts. First, we measured how long RapSheet takes to generate the



(a) Response times to generate provenance graphs for all alerts.



(b) Response times to generate TPGs with their threat scores.

Figure 3.12: CDF of response times to run RapSheet analysis.

provenance graph for each alert in our 58,096 alerts dataset. These provenance graphs are generated by performing backward and forward tracing queries for each alert, which reads the provenance graph database from disk. Figure 3.12a shows the cumulative distribution function (CDF) of response times for all the alerts. The results show that for 80% of alerts, RapSheet generates the provenance graph in less than 10 secs. Note that most of this time was spent in disk reads, which we can likely speed up using existing main-memory graph databases [96, 97].

Second, we measured the response time for performing tactical provenance analysis, which includes first extracting the IIP graph from the provenance graph of each alert, transforming this IIP vertex graph into a TPG, and finally assigning threat score to the TPG. For this response time, we assume that the provenance graph of the alert (from Figure 3.12a) is already in the main memory. Figure 3.12b shows that RapSheet was able to perform tactical provenance analysis and calculate threat scores on 95% of all the alerts in less than 1 ms.

3.5.6 Graph Reduction

To answer RQ3, we measured the graph size reduction from applying the technique discussed in Section 3.4. Figure 3.13 shows the percentage reduction in the number of edges for the 34 hosts in our evaluation, one bar for each host. On average, RapSheet reduces the graph size by 63%, increasing log buffer capacities by 2.7 times. Note that we saw a similar reduction in the number of vertices. In other words, the same end host can store 2.7 times more data without affecting storage capacity provided by EDR and data processing efficiency. This shows that skeleton graphs can effectively reduce log overhead.

Since currently RapSheet does not support cross-machine provenance tracking, our graph reduction algorithm is limited to ensure the correctness of causality analysis. Recall that

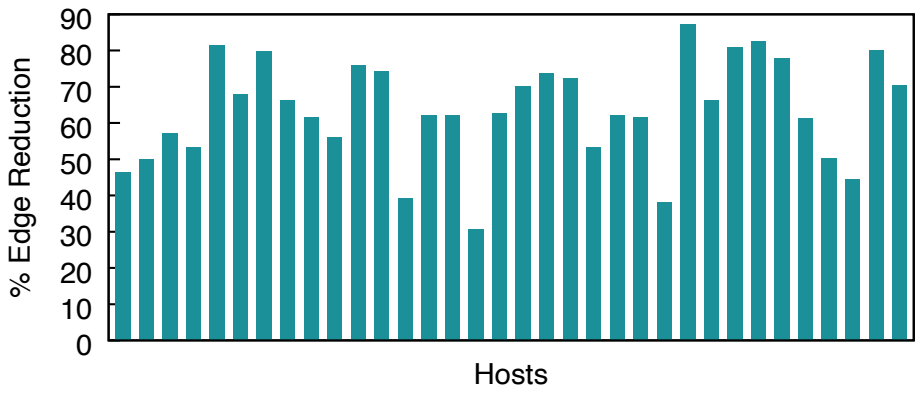


Figure 3.13: Percentage of edges removed from each host's provenance graph after applying our graph reduction algorithm.

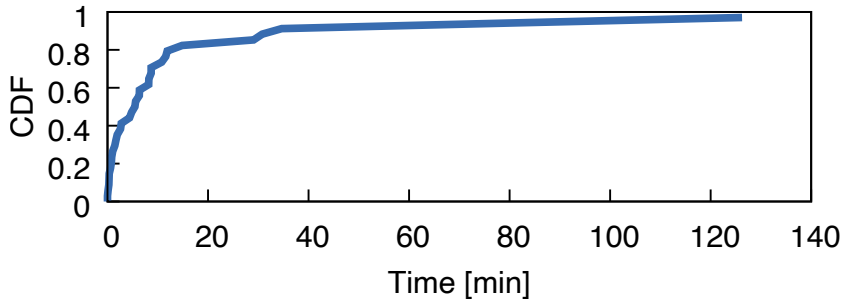


Figure 3.14: CDF of running graph reduction algorithm on each of the hosts' provenance graph.

our reduction algorithm does not remove a provenance path if it leads to some alert. So in our implementation we conservatively assume all the network connections made to hosts within our enterprise can lead to an alert and thus do not remove such network connections during the reduction process. We expect to see a further reduction in graph size once we incorporate cross-machine provenance analysis in the future work.

We also measured the cost of running our graph reduction algorithm on the full provenance graphs for the full duration of our data collection for each machine. The results are shown in Figure 3.14. As we can see, graph reduction finished in under 15 minutes on 80% of the hosts. In the worst case, one host took around two hours to finish. Upon further investigation, we found that this host has the highest number of edges in our dataset with 1.5M edges while the average is 370K edges. This overhead, which can be scheduled at times when machines are not busy, is acceptable for enterprises since the benefit of extra storage space from pruning graph while maintaining alert scoring and correlation outweighs the cost of running the graph reduction algorithm.

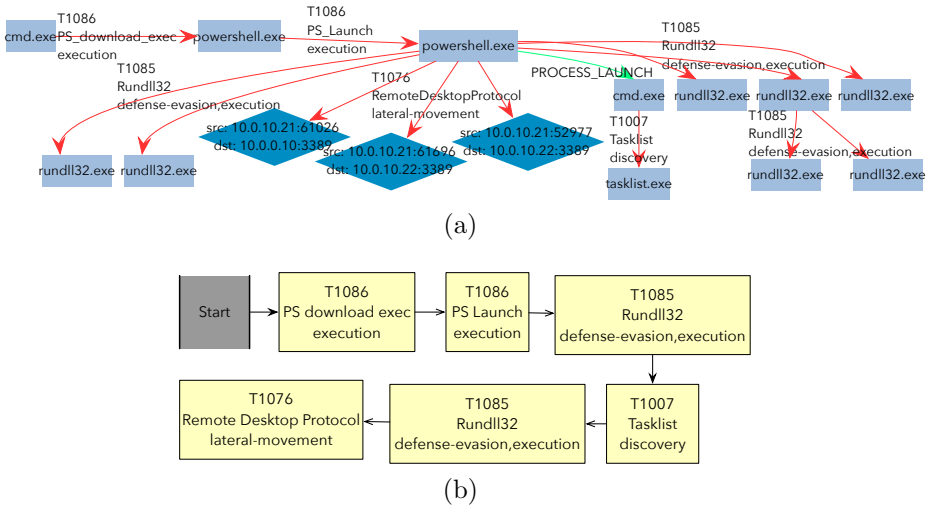


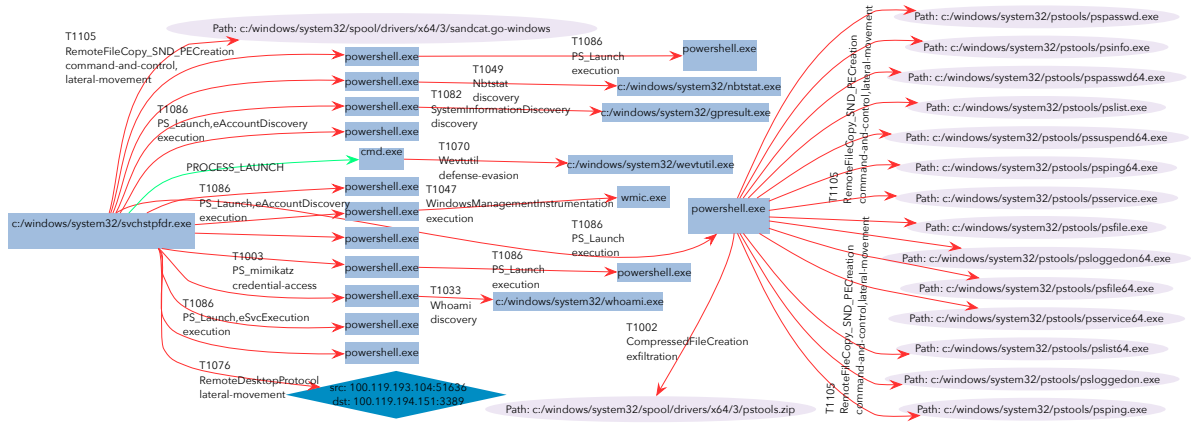
Figure 3.15: APT3 Attack Scenario. **(a)** IIP Vertex graph generated by RapSheet. **(b)** Tactical Provenance Graph for APT3 attack after applying readability post-processing pass. TPG is three orders of magnitude smaller than classical provenance graph. RapSheet will choose the maximum ordered tactic sequence from this TPG for the final threat score assignment.

3.5.7 APT Attack Campaign Case Studies

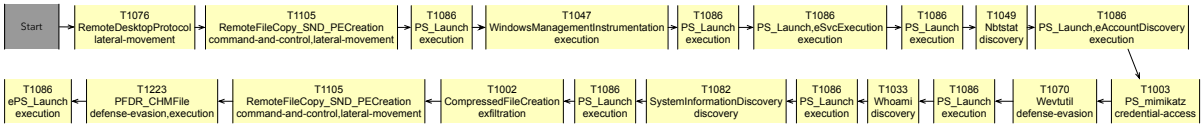
For our evaluation, we analyzed APT attacks from two well-known threat groups (APT3 and APT29) and one custom-designed attack executed using the MITRE CALDERA framework [98]. We already presented the APT29 attack scenario as a motivating example in Section 3.1. We now describe the APT3 attack and an APT attack using CALDERA.

APT3 is a China-based threat group that researchers have attributed to China's Ministry of State Security. This group is responsible for the campaigns known as Operation Clandestine Fox, Operation Clandestine Wolf, and Operation Double Tap [99]. Similar to APT29, APT3 has been well studied. ATP3's goals have been modeled using MITRE tactics and techniques. In our attack scenario, we performed various techniques from this known set ranging from System Service Discovery (T1007) to Remote Desktop Protocol (T1076). These techniques allowed us to achieve several of the MITRE tactics including execution, lateral movement, and defense evasion on the victim host. Figure 3.15a shows the IIP graph for the APT3 attack scenario, while Figure 3.15b shows the TPG extracted from this IIP graph. Our threat scoring algorithm ranked this TPG at number 15 out of 681, higher than the vast majority of the 676 false TPGs. To score this TPG, RapSheet found the following temporally ordered sequence of tactics: execution, defense-evasion, discovery, and lateral-movement.

In addition to the two simulations of actual advanced adversary groups that were per-



(a)



(b)

Figure 3.16: Caldera attack scenario. **(a)** IIP Vertex graph generated by RapSheet. We have omitted some of the edges and vertices from the graph for presentation. Complete IIP graph consists of 49 edges and 50 vertices. **(b)** Tactical Provenance Graph for Caldera attack after applying readability pass. RapSheet will choose the maximum ordered tactic sequence from this TPG for the final threat score assignment.

formed by red teams, we performed our own third campaign using a configurable, automated attack emulation framework called CALDERA [98] which is maintained by MITRE. CALDERA provides a client "malware" agent and a command-and-control server that agents can communicate with to receive commands to execute on the infected machines.

As a first step, we manually installed the client agent on one machine. This is a realistic scenario, since the initial infection stage is often missed by deployed defenses, either because a zero-day vulnerability is exploited or because the agent is installed by an unsuspecting, legitimate user. We then configured the command-and-control server to issue commands to discover other machines on the network, attempt to log in to those machines using stolen credential, copy the agent to any machines it successfully logged into, search for document files on all infected machines, zip up any found files and exfiltrate the files by sending the stolen file archives to the command and control server.

This covers a variety of ATT&CK techniques, from System User Discovery (T1033) to Remote File Copy (T1105), and several tactics including credential access, lateral movement, and exfiltration on the victim hosts. The IIP vertex graph and tactical provenance graph for one of the victim machines are shown in Figure 3.16.

CHAPTER 4: SWIFT: LIGHTNING-FAST DATA PROVENANCE ANALYTICS

Modern organizational networks are sprawling and diverse, hosting data of tremendous value to malicious actors. Unfortunately, due to the complexity of organizations and time-consuming nature of threat investigations, attackers are able to dwell on the target system for longer periods. In slow-moving targeted attacks (e.g., Equifax [2]), the amount of damage wrought by the attacker grows exponentially as their dwell time in the system increases [8], with a recent study reporting that it costs organizations $32,000 for each day an attacker persists in the network [100]. This situation is made even worse when considering fast-spreading attacks; the infamous Slammer worm [101] that infected more than 75,000 hosts within the first ten minutes of its release, and recent ransomware attacks [44, 102, 103] exhibit a similar replication factor. Regardless of the specific attack, delayed response times imply significantly larger negative consequences. Thus, to minimize repercussions of intrusions, cyber analysts require tools that facilitate fast and interactive threat hunting.

Given its vital importance, what are the key factors that determine the success of the threat hunting process? The various steps involved in post-breach threat hunting [104] are summarized in Figure 4.1. Effectiveness is usually measured using two metrics in industry [105]: 1) *Mean-time-to-detect* (MTTD), which measures the time required for the organization's Threat Detection Software (TDS) to detect suspicious activity and raise a security alert; and 2) *Mean-time-to-know* (MTTK), which measures the time required for cyber analysts to make sense of alert and unearth evidence that the alert is indicative of a true attack. Depending upon the volume of threat alerts and the analysis tools available to the analyst, this process can typically range from hours to days for an individual threat alert [4, 8].

Recently, threat hunting has become a subject of renewed interest in the literature, primarily due to advancements in *causal analysis* [10, 11, 12, 13, 14, 15, 16, 21, 33, 43, 106, 107, 108, 109, 110] that can can reduce MTTK during the post-breach threat hunting process. Unfortunately, at present the performance of causal analysis is a limiting factor to their widespread adoption – early attempts to deploy these techniques in practice reported graph construction times ranging from hours to days and unwieldy audit logs that reached terabytes in size over just a week (e.g., [20]). These existing tools fall under two categories: 1) disk-based offline approaches (e.g., [20, 30]) that incur significant I/O bottleneck and takes hours to respond to each query, thereby increasing MTTK; and 2) memory-based online approaches (e.g., [10, 11]) that require the *whole* causal graph to be stored in main-memory for analysis, which cannot scale to even modestly-sized organizations. As neither

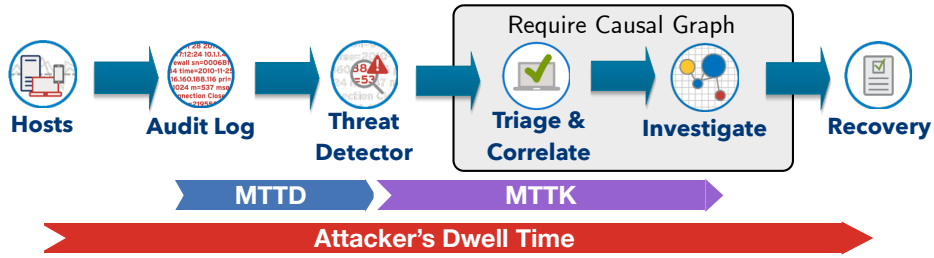


Figure 4.1: Typical post-breach threat hunting in an enterprise. Both alert management (e.g., triage) and investigation steps require causal graphs of generated alerts.

approach is a practical candidate for deployment, prior work has sought to improve the performance of causal analysis through various forms of graph reduction and compression (e.g., [18, 19, 22, 24, 25, 31, 111, 112]). By reducing the number of log events to process, those techniques have indeed improved query latency and alleviated the burdens of long-term storage. However, these approaches potentially affect the fidelity of logs for answering key forensic queries. For example, LogGC removes subgraphs associated with closed sockets and thus could obscure data exfiltration attempts [19], while Winnower may prevent attack attribution by abstracting remote IP addresses [18]. Further, over longer periods those techniques do not provide a scalable solution to log analysis and management.

4.0.1 Our Approach & Contributions

In this chapter, we propose a causal analysis and alert management framework that can process logs and forensic queries as quickly as the system event stream. Unfortunately, building a highly scalable real-time causality tracker is a daunting task. The challenge comes from the *volume* and *velocity* of system events that are in large enterprises. Three key challenges need to be answered before we can build this scalable mechanism:

C1 *Scalable Ingest:* How can we continuously ingest and process upwards of terabytes of system events per day?

C2 *Fast Graph Retrieval:* How can we quickly recover causal graphs of recent alerts, especially when alerts' dependencies may extend back weeks into the past?

C3 *Efficient Alert Management:* How can we incorporate causality analysis into real-time alert management to help cyber analysts cope with the deluge of alerts?

To address these challenges, we designed SWIFT[1], a causality tracker for which scalability

[1]SWIFT is a recursive acronym for Swift investigator for threat alerts.

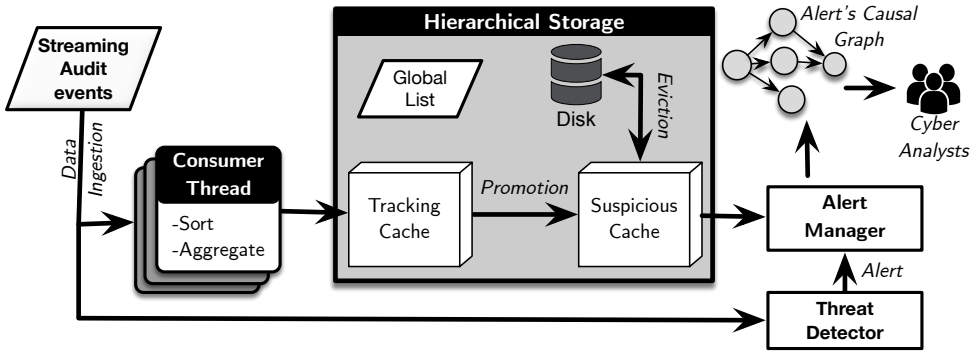


Figure 4.2: Overview of SWIFT architecture.

and performance are first-class citizens. Figure 4.2 presents an overview of the SWIFT architecture. Enterprise-wide audit logs are first collected into a Kafka broker [113] and then fetched by the consumer threads of SWIFT. Each consumer thread buffers the events for a certain configurable window, organizes the out-of-order events based on their timestamps, and merges continuous events that have the same source and destination.[2] Then, these audit log events are fed into a novel hierarchical storage management (HSM) system.

The challenge of *scalable ingestion* (C1) is met by the first contribution of this paper, a novel vertex-centric graph schema and database that is tailored for online causality analysis. This in-memory causal graph database allows SWIFT to quickly identify the causal relationships of streaming events with all causally-related events that occurred previously. We show that our graph database is space-efficient and is an enabling factor in providing real-time query results without significant disk I/O during our experiments.

The challenge of *fast graph retrieval* (C2) is resolved through the introduction of a causal graph HSM that consists of a two-layered memory cache (the tracking cache and suspicious cache, respectively), and a disk. This HSM automatically moves causal graph segments between main-memory and disk to achieve high-throughput data ingestion and low-latency query results. However, incorporating an HSM into an existing causal analysis framework is non-trivial – a generic cache eviction strategy would regularly evict forensically-relevant events, leading to increased disk access and high query latency.

Our solution to eviction is based on two distinct insights that motivate our two-layered memory cache design. The first insight is that of temporal locality; recent events have a high probability of dependence with upcoming events in the near future, Based on this observation we formulated an *Epochal Causality Hypothesis*, described in Section 4.3.1, and store recent events in the tracking cache. As events age out of the tracking cache, a decision

[2]Most of the operating systems introduce several system-level events for single file operation. Aggregating these events together does not affect the correctness of causality analysis but saves substantial space.

must be made as to which events are likely to be used in forensic queries and should thus be retained in memory.

To identify forensically-relevant events, we formulated a *Most Suspicious Causal Paths Hypothesis* which states that, given a suspicious influence score algorithm (e.g., [20, 30, 114, 115, 116]) that satisfies three key properties described in Section 4.3.2, we can calculate the most suspicious causal paths in an online fashion (on time-evolving graphs); as these paths are more likely to be associated with a true attack, they are also the most likely to be queried and should thus be retained in the suspicious cache. Note that a causal graph consists of one or more causal paths (further described in Section 4.1). Finally, to quickly identify top-k most suspicious causal paths seen so far in the enterprise, SWIFT also maintains a Global List that stores pointers to such paths.

The final contribution of this paper considers the matter of efficient alert management (C3), which is a vital consideration to mitigating threat alert fatigue [8]. SWIFT includes an alert management layer on top of its HSM. When alerts are fired by a connected TDS (e.g., Splunk [9]), SWIFT automatically leverages its suspicious influence scores to perform alert triage based on historical context,[3] allowing the analyst to investigate the most likely threats first. Further, during online causality tracking, SWIFT keeps track of all previously-fired alerts. When an alert has a causal relation with a previously fired alert, SWIFT fuses these events into a single causal graph to display to the analyst.

4.0.2 Summary of Results

We deployed and evaluated our system at NEC Labs America, comprised of 191 hosts. Our case studies on this testbed confirm that SWIFT can retrieve the most critical parts of an APT attack from a database of over 300 million events in just 20 ms. SWIFT successfully classified 140 security alerts and responded to forensic queries in less than 2 minutes, reducing the latency of the state-of-the-art alert triage tools by 5 hours. With this result, we estimate that SWIFT can scale to monitor upwards of 4,000 hosts on a single server. Further, SWIFT can scale to support thousands of monitored hosts on a single machine using just 300 MB memory, thus addressing a central limitation of existing causal analysis techniques. *We clarify at the outset that* SWIFT *does not improve or detract from the efficacy of its two modular components, the underlying TDS (e.g., [9]) and suspicious influence scoring algorithm (e.g., [20, 30]); instead,* SWIFT *seeks to improve security by dramatically improving the speed and scalability of causality-based threat hunting solutions.*

[3] Prior work [30] has shown that incorporating historical context into alert triage may reduce the false positives of a commercial TDS by up to 84%.

4.1 PRELIMINARIES

4.1.1 Causal Path

A causal graph consists of one or more causal paths, which are defined as follows:

Definition 4.1. Causal Path. A causal path P of a event e_a represents a chain of events that led to e_a and chain of events induced by e_a in the future. It is a temporally ordered sequence of events and represented as $P := \{e_1, \ldots, e_a, \ldots, e_n\}$ of length n. Each event can have multiple causal paths where each path represents one possible flow of information through e_a.

4.1.2 Suspicious Influence Score

When analyzing causal paths, it is desirable to understand how the suspiciousness of each event relates to the whole. Here, our *suspicion* may relate purely to an event's rarity, but may also incorporate other knowledge sources besides frequency, such as IP blacklists or antivirus signatures. To evaluate the suspiciousness of an entire path, we introduce the notion of a suspicious influence score. We say that a path exerts "suspicious influence" because it influences the level of suspicion that we have for future events, including alert events.

Definition 4.2. Suspicious Influence Score. For a causal path $P := \{e_1,\ldots,e_i,\ldots,e_n\}$ where the suspiciousness score for event e_i is given by $AS(e_i)$, the suspicious influence score $AS(P)$ is a function that combines the suspiciousness score of each event in the path P.

Many prior works satisfy this definition for a suspicious influence scoring algorithm, e.g., [20, 30, 114, 115, 116]. In our approach, we require the scoring algorithm to satisfy three specific properties: Cumulativity, Temporality, and Monotonicity. Combined, these properties will allow SWIFT to track causality in an online fashion with a low time complexity and minimal disk operations. To better explain these properties, we use Figure 4.3 as an example.

The first property, **Cumulativity**, means that the suspicious influence score of a path can be calculated from the suspicious influence score of its prefix and the suspiciousness score of its last event. For example, in Figure 4.3, to calculate the suspicious influence score of the causal path $P_1 = \{B \to A \to D\}$, we only need to know the suspicious influence score of $P_1' = \{B \to A\}$ and the suspiciousness score of event $A \to D$. This property guarantees

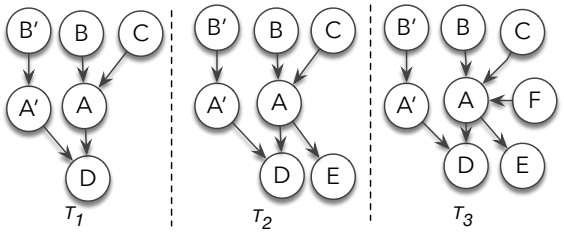


Figure 4.3: Example of causal graph database updates over time.

that while adding new events to an existing path, SWIFT does not need to backtrack the existing path to generate the suspicious influence score for the newly extended path.

The second property, **Temporality**, means that an event can only affect the suspicious influence score of events that happen after it. For two events $e_1 = \{V_1 \rightarrow V_2\}$ and $e_2 = \{V_2 \rightarrow V_3\}$, event $AS(e_2)$ depends on $AS(e_1)$ only if e_1 happens before e_2. This is intuitive from an information flow perspective, as V_2 will not have been inform by V_1 until after e_1 occurs. For example, at time T_2 in Figure 4.3, events $A \rightarrow E$ and $A \rightarrow D$ do not depend on event $F \rightarrow A$ because this occurred at time T_3. Therefore, we do not calculate the suspicious influence scores $AS(F \rightarrow A \rightarrow E)$ or $AS(F \rightarrow A \rightarrow D)$.

The third property **Monotonicity**, means that when a new event is appended to two existing paths it does not change the suspicious influence score of the existing paths. Let $P_1 = \{P_1' \rightarrow S \rightarrow D\}$ and $P_2 = \{P_2' \rightarrow S \rightarrow D\}$, where P_1' and P_2' are distinct causal paths prefixes and $\{S \rightarrow D\}$ is a new event shared by P_1 and P_2. The monotonicity property states that if $AS(P_1' \rightarrow S) > AS(P_2' \rightarrow S)$ then it must also be true that $AS(P_1) > AS(P_2)$. For example in Figure 4.3, if $AS(B \rightarrow A) > AS(C \rightarrow A)$ at time T_1 then it must also be true that $AS(B \rightarrow A \rightarrow E) > AS(C \rightarrow A \rightarrow E)$ at time T_2. This property helps ensure the correctness of our online causality tracking.

4.2 VERTEX-CENTRIC CAUSAL GRAPH

In this section, we first explain different graph formats and describe their merits and limitations for fast causal analysis. Then, we present the graph format used by SWIFT.

4.2.1 Graph Representation

There are two major data formats for graphs [117]. First, the *Edge List* format is a collection of edges, each a pair of vertices, that captures the incoming data in their arrival order. Second, the *Adjacency List* format manages the neighbors of each vertex in separate

per-vertex edge arrays. In Edge Lists, the neighbors for each vertex are scattered across the data structure, making it difficult to traverse the graph quickly. On the other hand, in Adjacency Lists vertex neighbors are easy to reference, making them better suited for causal graph traversal.

In our causal graph schema, each system subject and object is represented as a vertex in the causal graph and stored as an entry in a key-value storage. In each key-value pair $\langle Key, Val \rangle$, Key is the unique identifier representing the vertex and Val is a list of three entries. For a vertex K this list is as follows:

1. A list of K's parent vertices' unique identifiers, $L_{parents}$. Each parent identifier is associated with a timestamp for the event's creation and the edge relationship type.

2. A list of K's child vertices' unique identifiers $L_{children}$. Each child identifier is associated with a timestamp for the event's creation and the edge relationship type.

3. An ordered list $PATH_{abnormal}$ of the m most suspicious causal paths that end with vertex K, sorted in order of each path's suspicious influence score.

This graph representation is specifically tailored towards forensic analysis queries, i.e., backward and forward tracing queries. We use the same graph representation for both main-memory and on-disk storage. Recall that a major goal of SWIFT is to provide hierarchical storage that can quickly query the most suspicious causal graphs. Our graph schema supports this through the $PATH_{abnormal}$ objects, which are sorted in a descending order of their suspicious influence scores. Note that each vertex has a set of causal paths that end at it, even though these may be sub-paths of other paths. For example, in Figure 4.3, vertex A has two paths in its $PATH_{abnormal}$, $P_1 = \{B \rightarrow A\}$ and $P_2 = \{C \rightarrow A\}$. These two paths are the sub-paths of $P_3 = \{B \rightarrow A \rightarrow D\}$ and $P_4 = \{C \rightarrow A \rightarrow D\}$, respectively.

4.2.2 Suspicious Causal Paths

For a vertex K, each path in $PATH_{abnormal}$ is a tuple in the form of $(P, S, t, Rel, Rank)$: P is the unique identifier of the parent vertex of K in a given causal path; S is the suspicious influence score of the path; t is the timestamp of the edge event $P \rightarrow K$; Rel is edge relationship between K and P; and $Rank$ is the relative score ranking of all the paths that end at $P \rightarrow K$. In the case when multiple edges with the same edge relationship Rel exists between two vertices, we keep only the latest timestamp. This is because ignoring the previous edges does not affect the correctness of forensic analysis, as shown by previous works (e.g., [11, 19]).

Algorithm 4.1: PATHDISCOVER

```
   Inputs : V, R, Seen
   Output: PATH
 1 Parent = GETPARENT(V,R)
 2 if Parent = Null then
 3 |   return Null
 4 if Parent ∈ Seen then
 5 |   return Null
 6 Seen ← Parent
 7 ParentRank = GETRANK(V,R)
 8 PATH = PATHDISCOVER(Parent,ParentRank, Seen)
 9 APPEND(PATH,Parent)
10 return PATH
```

We use Figure 4.3 as an example to explain our design of $PATH_{abnormal}$. Note that we do not show edge relationships in this figure and rest of the paper for simplicity although we do store edge relationships in our schema. In Figure 4.3 there are three paths ending at the vertex D, which are $P_1 = \{B \rightarrow A \rightarrow D\}$, $P_2 = \{C \rightarrow A \rightarrow D\}$, and $P_3 = \{B' \rightarrow A' \rightarrow D\}$. Assume the suspicious influence score and the timestamps of P_1, P_2, and P_3 are S_1, S_2, and S_3 and t_1, t_2, t_3, respectively. If $S_1 > S_2 > S_3$, then $PATH_{abnormal}$ of D is $[(A, S_1, t_1, Rel_1 0), (A, S_2, t_2, Rel_2, 1), (A', S_3, t_3, Rel_3, 0)]$. For the tuple $(A, S_1, t_1, Rel_1, 0)$, it means that the parent of the given causal path is A, its suspicious influence score is S_1, the event $A \rightarrow D$ happens at time t_1 with edge relationship Rel_1 and its suspicious score ranks the first among all paths which have the last edge as $A \rightarrow D$.

Limiting the Size of $PATH_{abnormal}$ The number of paths that end at each vertex is exponential to the number of vertices. Maintaining a $PATH_{abnormal}$ that contains all paths is not realistic. To address this limitation, in our design of SWIFT, the length of $PATH_{abnormal}$ is limited to m. Limiting the size of $PATH_{abnormal}$ means that for each vertex in the causal graph, SWIFT only keeps the top m most suspicious paths that end at that vertex in memory. Note that this does not affect the completeness of the whole causal graph since the complete parent and child list for each vertex is maintained on disk. It only affects the paths that can be retrieved quickly from the main memory. Based on the Hypothesis H2, these suspicious paths are more likely to represent attacks. Thus, it is reasonable for us to limit the size of $PATH_{abnormal}$ for each vertex.

4.2.3 Graph Query

Our design of the causal graph schema and database allows fast recovery of a causal path with the unique identifier of its last vertex and its index in $PATH_{abnormal}$. The time

complexity of the recovering process is $O(n)$, where n is the length of the causal path. The algorithm is outlined in Algorithm 4.1. The inputs are the vertex V, the relative ranking R of the causal path in V, and $Seen$, which is a hashmap of the previously-visited vertices during path discovery. This hashmap is used to halt recursion in the case of a cycle. The output of Algorithm 4.1 is the discovered path.

We use Figure 4.3 as an example to explain the recovering process. Assume that SWIFT wants to recover the highest scoring path $P_1 = \{B \rightarrow A \rightarrow D\}$. To do so, SWIFT only needs to have the last vertex D and the relative ranking (index), which is 0. To recover the full path, SWIFT refers to the first element in its $PATH_{abnormal}$ and recovers the parent in the given path, which is A, and gets the relative ranking of the path in A, which is also 0. Then this process is recursively repeated on A and its ranking until the whole path is recovered.

4.3 HIERARCHICAL STORAGE AND ALERT MANAGEMENT

4.3.1 Tracking Cache

SWIFT takes the stream of audit log events and identifies causal relationships between each new event and past events in order to build a causal graph. The role of the tracking cache is to ensure that the events most relevant to the graph building process are consistently available in the main memory. Our approach to assuring fast access to causally-related past events is based on the following hypothesis:

H1 Epochal Causality Hypothesis. *Events which are recently accessed during causal graph generation are accessed again in a short epoch of time ($\Delta T_{promote}$), and thus should not be evicted from the main memory in that epoch.*

An empirical validation of hypothesis H1 is given in Figure 4.4 based on the audit stream of a 191 host enterprise. This CDF shows that the immediate dependencies (i.e., parents) of 98% of newly created events were created within a short epoch prior (< 15 mins). In other words, if we can design a cache that can store the most recent 15 minutes of events in the main memory, we will eliminate 98% of disk accesses.

Tracking Algorithm & Eviction Policy. Algorithm 4.2 outlines the high level steps of our online tracking algorithm. At the high level, it takes the causal graph database (GDB) and an incoming event (E) as the input, and adds E's subject and object to the causal graph database as two vertices. At the same time, the algorithm calculates and updates $PATH_{abnormal}$, which represents the most suspicious paths that end with the object of E. The time complexity of updating the $PATH_{abnormal}$ is O(1).

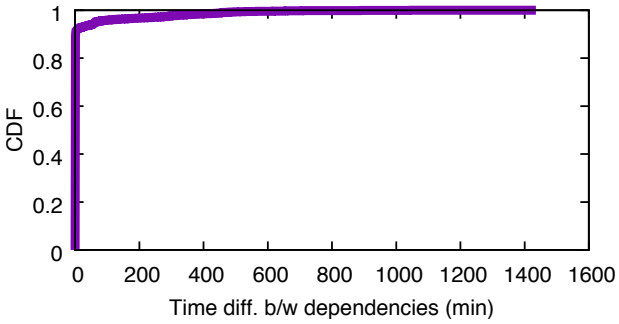


Figure 4.4: CDF of the time difference between a newly generated event and the event's immediate dependencies (i.e., parents). 98% of events' immediate dependencies occurred less than 15 minutes ago, providing empirical evidence for the Epochal Causality Hypothesis.

The first step of the tracking algorithm is to check if the subject and the object of the event exist in the GDB (lines 1-2). RetreiveOrCreate does this work. Given the system entities (the subject or the object), RetreiveOrCreate tries to first fetch it from the main memory. If the system entity does not exist in the main memory, RetreiveOrCreate tries to fetch it from the disk. If the system entity still does not exist in the hard disk, RetreiveOrCreate will create a new entry in the causal graph database.

Once the subject and object have been retrieved, Swift updates the parent and child list for the subject and object based on edge relationship Rel (lines 4 - 5 and lines 12 - 13). Then it updates the $PATH_{abnormal}$ list of the children (lines 6 - 9 and lines 16 - 19). To do so, Swift enumerates each element in the $PATH_{abnormal}$ of the subject (line 6), calculates a score from each element in the subject's $PATH_{abnormal}$ (line 7) and updates the $PATH_{abnormal}$ of the object with the new score, the relative ranking the in subject ($Index$), and the new time stamp of the event (line 8). Finally, Swift updates the GDB of the subject and the object in the main memory.

The time complexity of Algorithm 4.2 is $\mathcal{O}(1)$. Since we have limited the size of the $PATH_{abnormal}$ as a constant, the time complexity of the loop between line 5 and line 8 is constant. Due to the same reason, the time complexity of AddToPath is also $\mathcal{O}(1)$. After each epoch $\Delta T_{promote}$, Swift evicts system objects (vertices) from tracking cache to the suspicious cache if they have not been accessed in the last epoch. Vertices that have been accessed during the past epoch are retained in the tracking cache for the next epoch.

4.3.2 Suspicious Cache

After being evicted from the tracking cache, vertex entries are moved to the suspicious cache. The goal of the second cache is to retain vertex entries for all vertices that fall on

Algorithm 4.2: TrackObject

```
   Inputs : GDB, E
 1 Sub = RetreiveOrCreate(E.sub, GDB)
 2 Obj = RetreiveOrCreate(E.obj, GDB)
 3 if IsParent(Sub, E.Rel) then
 4 |   Sub.AddChild(Obj)
 5 |   Obj.AddParent(Sub)
 6 |   for Index, (P, S, t, R) ∈ Sub.PATH_abnormal do
 7 |   |   ChildScore = CalculateScore(S,Sub, Obj, E)
 8 |   |   Obj.AddToPath(Sub, ChildScore, E.t, Index)
 9 else
10 |   Obj.AddChild(Sub)
11 |   Sub.AddParent(Obj)
12 |   for Index, (P, S, t, R) ∈ Obj.PATH_abnormal do
13 |   |   ChildScore = CalculateScore(S,Sub, Obj, E)
14 |   |   Sub.AddToPath(Obj, ChildScore, E.t, Index)
15 GDB.Update(Sub)
16 GDB.Update(Obj)
17 return
```

the Top K most suspicious causal paths throughout the history of system execution. The intuition behind the suspicious cache is based on the Hypothesis H2.

H2 Most Suspicious Causal Paths Hypothesis. *If a path in the causal graph contains multiple suspicious (anomalous) events, it is much more likely to be associated with a true attack.*

Recent studies provide evidence for this hypothesis, and in fact are the inspiration for the present study – Hassan et al. [30] present an alert triage system that ranks alerts based on the aggregate anomalousness of their causal paths, observing that this approach can be used to eliminate 84% of false alerts from a commercial Threat Detection Softwares (TDS). Liu et al. [20] present an optimization for forward trace queries that prioritizes the search of anomalous paths in order to construct attack graphs more quickly. While these results are encouraging, both of these systems rely on disk-based graph storage and are thus subject to extremely high latencies when traversing causal graphs; our observation is that this hypothesis can also inform the design of a forensic cache. Because true attacks are likely to fall on the most suspicious (anomalous) causal paths, our system should prioritize the retention of events associated with such paths. This will increase the likelihood that all forensically-relevant information will exist in main memory at the time of the investigation.

The specific goal of the suspicious cache is to retain vertices that appear in the Top K most suspicious causal paths; we call this set of K paths the Global List (GL). Each element in Global List is a pair (V, R), where V is a vertex in the causal graph database and R is the index of the causal path in V's $PATH_{abnormal}$ list. As discussed in Section 4.2, we can recover the full causal path efficiently with this pair.

For a causal path P to be in GL, P must meet three conditions: (1) the suspicious influence score of P is among the top K most suspicious paths in history; (2) P is not a sub-path of another causal path (e.g., the path $\{B \rightarrow A\}$ in Figure 4.3 could not be in GL because it is a sub-path of $\{B \rightarrow A \rightarrow D\}$; and (3) P is in the $PATH_{abnormal}$ of at least one *vertex*. The third condition alleviates a possible "spoofing attack" that spoils the cache of SWIFT, which we discuss in Section 4.5.

Suspicious Cache Eviction Policy. Based on the Hypothesis H2, SWIFT maintains the top-K most suspicious causal paths in the memory to support low-latency attack investigation. To achieve this goal with our two-layer cache design, we introduce a time-window ΔT_{evict} to evict objects from suspicious cache and GL to the disk. At a pre-defined time interval, ΔT_{evict}, SWIFT asynchronously runs an eviction algorithm to move the vertices that are not contained in a GL path to the disk. Algorithm 4.3 outlines the high-level steps of the eviction process from suspicious cache to the disk. Its inputs are GL and suspicious cache. The algorithm first enumerates every tuple in GL, recovering the causal path from the tuple using Algorithm 4.1. Then, for each vertex in the recovered path, it taints the vertex as "TO_KEEP" (lines 1 - 5). After the tainting process, SWIFT evicts the key-value pairs in the *SuspiciousCache* that do not have the "TO_KEEP" taint (lines 6 - 11). Note that Algorithm 4.1 accounts for possible cycles in the graph.

Correctness. The equation we use for suspicious influence scoring in SWIFT's implementation is given by Equation 4.1 in Section 4.4. Based on the Cumulativity this equation, the time complexity of CALCULATESCORE is also $\mathcal{O}(1)$. The correctness of Algorithm 4.2 is guaranteed by the Monotonicity and the Temporality of Equation 4.1. Due to the Temporality of Equation 4.1, Algorithm 4.2 only needs to update the $PATH_{abnormal}$ for the object. It does not need to further propagate the suspicious influence score to the successors. Due to the Monotonicity, the new top m most suspicious paths of the object can only be from the old $PATH_{abnormal}$ of the object or the new causal paths generated from the top m most suspicious paths of the subject. Thus, to calculate the new $PATH_{abnormal}$, it is safe to only enumerate the items in $PATH_{abnormal}$ of the subject.

Time Complexity. Our eviction algorithm runs in $\mathcal{O}(N)$ time complexity, where N is the total number of vertices present in the main-memory, since it has to taint all the vertices

Algorithm 4.3: EVICTION

```
   Inputs : GL, SuspicousCache
1  for (V, R) ∈ GL do
2  |  PATH= PATHDISCOVER(V, R)
3  |  for N ∈ PATH do
4  |  |  TAINT(N)
5  for < K, V > ∈ SuspiciousCache do
6  |  if CHECKNOTTAINT(< K, V >) then
7  |  |  EVICT(< K, V >)
8  return
```

which belong to the Global List path and evict the vertices that do not belong to the Global List path.

4.3.3 Alert Management

The alert management layer provides three fundamental capabilities: context-based alert triage, alert correlation, and suspicious causal graph generation. These capabilities are based on SWIFT's HSM which allows them to be real-time. Context-based alert triage is achieved by the propagating and storing of suspicious influence scores along with each causal path in the database. Note that the suspicious influence scores are calculated during online tracking (Algorithm 4.2). As discussed previously, the greater the suspicious influence score of an alert, the more suspicious that alert will be and should therefore be investigated first. As soon as alerts are fired during threat hunting process (shown in Figure 4.1), SWIFT iteratively sorts alerts based on suspicious influence scores. In the alert management stage, SWIFT only needs to retrieve the previously-calculated suspicious influence scores from the HSM, assuring that alert triage can occur in real-time.

Alert correlation and causal graph generation are realized automatically by our HSM. SWIFT uses the suspicious cache to retain the causal paths of those previously triggered alerts that have higher suspicious influence scores. To correlate two alerts, SWIFT only needs to query the suspicious cache to figure out if the most recently triggered alert's causal path is associated with any alerts that were triggered in the past. To support graph generation, SWIFT provides two types of queries to retrieve the graph of alerts: concise queries and complete queries. The concise query returns the most suspicious causal subgraph related to an alert, which is stored entirely in the suspicious cache. The complete query returns the whole graph by fetching paths from both the suspicious cache and, if needed, the disk.

4.4 EVALUATION

In this section, we focus on evaluating the efficacy, usefulness, and scalability of SWIFT as a real-time forensic analysis in an enterprise. In particular, we investigated the following research questions (RQs):

RQ1 How effective is SWIFT in threat alert investigation?

RQ2 What are the insights into the events that are cached vs spilled to disk by SWIFT?

RQ3 How scalable is SWIFT?

RQ4 Can the time saved using SWIFT help an enterprise to thwart an attack?

RQ5 How efficient is SWIFT at alert management?

4.4.1 Implementation

We implement SWIFT for an enterprise environment and collected system event logs generated by Windows ETW [55] and Linux Auditd [56] using Kafka producers. We wrote our own consumer threads to fetch audit logs from Kafka producers. SWIFT uses the Guava Cache by Google [118] to maintain the causal graph database in the main-memory. This cache supports timed eviction and asynchronous batch writes. SWIFT uses RocksDB [119] as the persistent key-value storage. The batch mode in RocksDB provides high rate for read and write.

In our implementation, we use the method proposed by Hassan et al. [30] to calculate the suspicious influence score because it satisfies all the three properties mentioned in Section 4.1.2. Particularly, for a causal path P, we calculate its Suspicious Influence Score $SIS(P)$ with Equation 4.1.

$$SIS(P) = 1 - \prod_{i=1}^{l} IN(SRC_i) \times M(\varepsilon_i) \times OUT(DST_i) \times \alpha \tag{4.1}$$

The details about the above-mentioned equation can be found in [30]. At a high-level, IN and OUT are two vectors that quantify the likelihood that the vertex is a source or destination of information flow, respectively. M is the transition probability from SRC_i vertex to DST_i vertex. α is a normalization factor. IN, OUT, M, and α are parameterized based on observations of historic benign data from the enterprise deployment. This equation satisfies all three properties mentioned in Section 4.1.2. *Cumulativity* is satisfied because this

Table 4.1: APT attack scenarios used in our evaluation with short their descriptions.

Attacks	Short Description
VPNFilter [63]	An attacker used known vulnerabilities [60] 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.
Redis-Server	Example case study in Section 4.4.2
wget-gcc [22]	Malicious source files were downloaded and then compiled.
WannaCry [44]	An attacker exploits EternalBlue [46] vulnerability in enterprise to gain access to machines and then attacker encrypts data on those machines.
Data Theft [20]	An attacker downloaded a malicious bash script on the data server and used it to exfiltrate all the confidential documents on the server.
ShellShock [60]	An attacker utilized an Apache server to trigger the Shellshock vulnerability in Bash multiple times.
Netcat Backdoor [61]	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
Cheating Student [21]	A student downloaded midterm scores from Apache and uploaded a modified version.
passwd-gzip-scp [22]	An attack stole user account information from passwd file, compressed it using gzip and transferred the data to a remote machine
Jeep-Cherokee [120]	An attack remotely exploits in-car information system and gains control over physical components (*e.g.*, wheels, breaks, engines) by sending out commands via CANBUS.

equation calculates score of each event by taking the product of all previous events' aggregate score and the new event's score. *Temporality* is preserved because the product is taken over a causal path, which is sorted temporally by definition. If a new event is added to two paths, the subtraction of their $SIS(P)$ will be multiplied by the same factors, which will not change their orders. Therefore, *monotonicity* is satisfied.

4.4.2 Experiment Setup

We collected system events and threat alerts at NEC Labs America. In total, we monitored 191 hosts (51 Linux and 140 Windows OS) for 10 days. We deployed SWIFT on a server with Intel® Xeon(R) CPU E5-2660 @ 2.20GHz and 64 GB memory running Ubuntu 16.04 OS. We connected SWIFT to ASI [36], a commercial anomaly-based TDS, to generate alerts. During the engagement, we injected 10 APT attacks over a period of 10 days. These APT attacks were designed by expert analysts employed at NEC Labs America. A short description of these attacks is shown in Table 4.1. On each day we injected one attack, except for 3 attacks (Datatheft, ShellShock, and Netcat backdoor) which were ran on the same day.

We collected more than 1 TB worth of audit logs with around 1 billion system events from 191 hosts over period of 10 days. The APT attack traces constitute less than 0.0005% of

the total audit logs collected from the enterprise. Meanwhile, we also monitored these logs with a commercial TDS [36]. This underlying TDS generated 140 threat alerts over a period of 10 days. Out of these 140 alerts, 12 were true alerts generated by our simulated APT attacks, while the rest were false alerts.

To evaluate SWIFT against a baseline approach, we re-implement NoDoze based on its description in [30]. We chose this as a baseline because it is one of the most recent offline approach that can perform: 1) suspicious score assignment, 2) automated alert triage and 3) causality graph generation. Further, our decision to implement SWIFT using NoDoze's suspicious influence scoring algorithm permits an apples-to-apples comparison when evaluating SWIFT's HSM. We used 20 consumer threads to consume audit logs from Kafka producers and then we performed forensic analysis in real-time. Note that 20 threads is also the maximum number of threads supported by the machine we use in our evaluation.

Parameters. We set $\Delta T_{promote} = 800$ *seconds*, GL size $K = 3000$, $PATH_{abnormal}$ size $m = K/3$, and $\Delta T_{evict} = 1600$ *seconds* in all experiments unless we explicitly note otherwise. We chose these values because they generate the optimal throughput and can hold all the suspicious data in our enterprise. However, we also discovered that it is flexible to choose the parameters for SWIFT since the throughput is not heavily affected by the value of the parameters.

RQ1: Effectiveness in Alert Investigation

To answer this question, we used SWIFT to generate the *most suspicious causal graph* for all 140 threat alerts, measuring the response time for answering each causal graph query. We issued each query at the end of the day, *not* immediately following the attack, which ensured: 1) all attack related events had been evicted from the tracking cache, and were thus either in the suspicious cache or on disk; 2) a steady state for the HSM where all promotion and eviction cycles were completed for that day. We manually verified the fidelity of SWIFT's causal graph for each alert against the graphs generated by the baseline approach, checking that SWIFT returned all of the critical events necessary to explain the attack.

The results for SWIFT are shown in Figure 4.5; SWIFT was able to respond in less than one second for 80% of the alerts because of our novel suspicion-based HSM. In total, SWIFT took less than two minutes to generate the concise causal graphs for all alerts. We compare these results to the baseline approach in Figure 4.6, noting that the scale on the x-axis has changed from seconds to minutes. It took more than 1 hour for the baseline approach to process the same set of alerts. Moreover, the baseline approach took more than three minutes for 40% of the alerts and more than 20 minutes for 25% of the alerts, *in the worst case taking more*

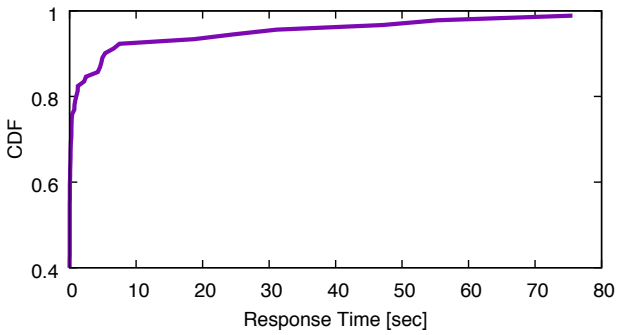


Figure 4.5: Response times *in seconds* to return concise causal graphs of threat alerts using SWIFT.

than an hour to finish. Such a slow response time is problematic, especially considering realistic scenarios in which the processing latency for one alert adds to the queuing latency of hundreds of other alerts in the stack (discussed more in **RQ3**).

A breakdown of performance results for each attack are shown in Table 4.2. The rightmost columns show the response time for the baseline method, SWIFT, and observed speedup. In all cases, SWIFT generated the causal graph for the attack in less than 3 milliseconds, whereas the baseline required nearly 5 minutes in the worst case. *Comparing the two techniques, we observe a speed up of up to 1.3 million times (Shellshock).* In spite of the performance increase, it may at first glance seem that the performance of the baseline approach is acceptable. One reason for this is that the underlying TDS used in our experiments itself maintained a 15GB event cache that was able to store part of the attack provenance for the baseline (compare this to the 300MB cache required by SWIFT, which we will show in Section 4.4.2). More importantly, a limitation of our evaluation is that it does not capture longitudinal attack patterns that are commonly observed in-the-wild, e.g., the 4.5 month attack window of the Equifax breach [2]. In such circumstances, the TDS cache would be useless and the baseline may take hours or days to process individual alerts.

Reasons for Milli-second Level Response Time. To further investigate the reason for the time reduction, we also studied what was maintained in the memory for these attacks. In Table 4.2, the column "All Events" represents all enterprise-wide system events collected while "Critical Events" represents only attack-related events. "%Cached" shows the percentage of events cached in the memory *end of the day.* Our experiment shows that SWIFT had a much lower response time because it effectively cached most of the events that were related to attacks in the main memory even if the size of the cache was small compared to the size of total events. Particular, on average, by maintaining about 0.04% of total events of a day, SWIFT can maintain on average 90% of attack-related events in its cache. In other words,

Table 4.2: Comparison of SWIFT's effectiveness against baseline. "#Alerts" shows how many alerts are associated with the particular attack.

Attacks	Reference	#Alerts	All Events		Critical Events		Response Time		
			Total	%Cached	Total	%Cached	Baseline	Swift	Speedup
VPNFilter	NoDoze [30]	1	150M	0.06%	15	100%	0.5 min	0.65 ms	46,000×
Redis-Server	Case Study Sec. 4.4.2	2	100M	0.07%	29	86%	1.1 min	0.1 ms	660,000×
wget	Xu et al. [22]	1	160M	0.03%	15	100%	0.7 min	1.1 ms	28,000×
WannaCry	NoDoze [30]	2	139M	0.05%	21	90%	1.5 min	0.2 ms	450,000×
Data Theft	PrioTracker [20]	1	366M	0.04%	13	88%	3 min	0.3 ms	600,000×
ShellShock	CVE-2014-6271	1	366M	0.04%	25	82%	2 min	0.09 ms	1,333,000×
Netcat Backdoor	Backdoor [61]	1	366M	0.04%	14	85%	1.8 min	0.8 ms	135,000×
Cheating Student	ProTracer [21]	1	336M	0.03%	37	94%	2.1 min	1.9 ms	66,000×
passwd-gzip-scp	Xu et al. [22]	1	335M	0.02%	25	90%	4.6 min	2.8 ms	98,000×
Jeep-Cherokee	Exploit Vehicle [120]	1	129M	0.06%	16	94%	1.2 min	1.2 ms	60,000×

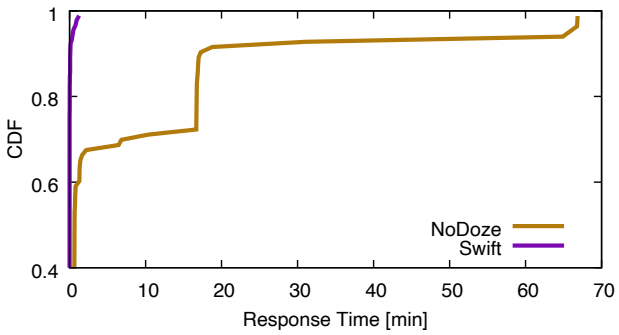


Figure 4.6: Response times *in minutes* to return concise causal graphs of threat alerts using SWIFT as compared to NoDoze (baseline). Note that the SWIFT CDF is the same as in Figure 4.5 on a different scale.

SWIFT was able to significantly reduce disk IOs while generating causal graphs for attacks. This result also validates our Hypothesis H2. Reasons for why SWIFT cannot maintain 100% of the attack-related events in its cache will be discussed in **RQ2**.

RQ2: Insights into Cached vs Spilled Events

To further study how causal events are handled in the SWIFT HSM (i.e. which events are cached, as opposed to being spilled to disk), we select a ransomware attack as a case study from the 10 attacks in Table 4.2. In this attack, a misconfigured Redis server [121] allows an attacker to log into the server via the `ssh` service as root [122]. The attacker first connects directly to a misconfigured Redis server over its default port, executes the `Flushall` command to erase the whole database, uploads their `ssh` key to the database, then obtains root access to the server by using `CONFIG` to copy the database to the root's `.ssh` directory and renaming it to `authorized_keys`. Once in the enterprise network, the attacker moves laterally in their search for valuable data while simultaneously encrypting data by running an `encryptor` that was downloaded from their remote server. *Time is crucial in this scenario – the earlier we investigate and respond to the attack, the more valuable data we can save.*

This attack generated two alerts which are marked in red dashed arrows in Figure 4.7. However, these true alerts are among a deluge of unrelated false alerts being generated by TDS, making it critical to quickly identify the true alerts and take actions to prevent damages. Fortunately, SWIFT assigns suspicious influence scores in real-time; when **Alert 1** arrives, SWIFT automatically remembers its suspiciousness score and propagates this score to its successors. When **Alert 2** fires, SWIFT combines the suspiciousness influence scores from **Alert 1** in $\mathcal{O}(1)$ time. This means that as soon as **Alert 2** is fired by TDS, SWIFT can instantaneously generates the most suspicious causal graph and correlate the alerts.

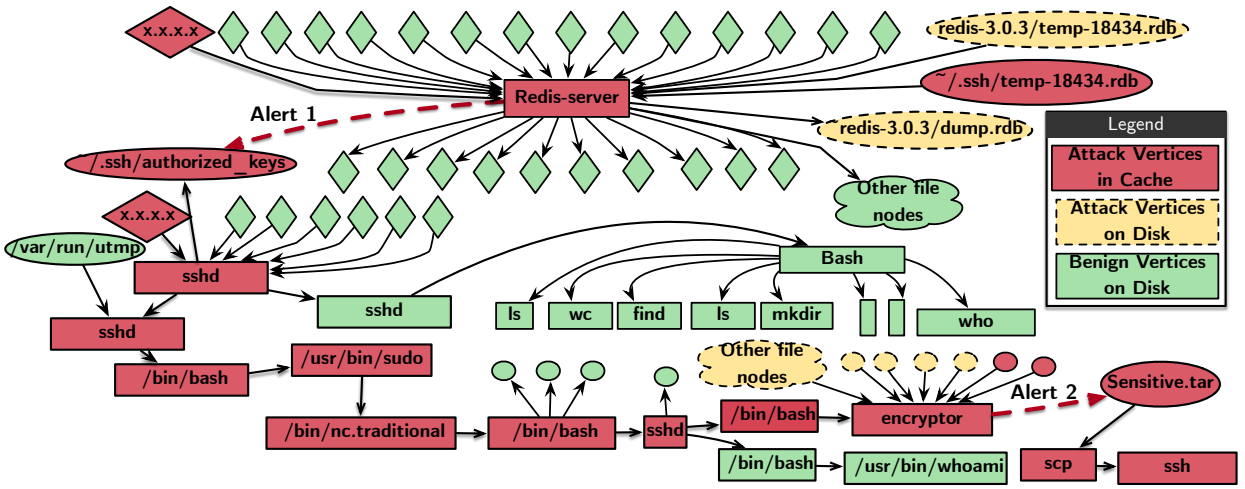


Figure 4.7: Simplified causal graph of the simulated ransomware attack. SWIFT keeps part of the causal graph related to the ransomware attack in the main-memory (red vertices), and part of that graph (yellow vertices) is spilled to the disk. Causal graph not related to the attack (green vertices) is spilled to the disk.

Figure 4.7 shows the simplified causal graph of this attack. In this graph, we use diamonds to represent sockets, oval nodes to represent files, and boxes to represent processes. In Figure 4.7, the red vertices represent the most suspicious causal graph which is cached in the main-memory. Yellow vertices are related to attack but spilled to disk while green vertices are not related to attack (benign) which are also spilled to disk. Due to dependency explosion problem (false dependency) [21] benign vertices become part of attack's causal graph. SWIFT shows the most suspicious graph (red vertices) to cyber analyst accelerate investigation and assist cyber analyst to *quickly* identify the root cause (`X.X.X.X` connection to process `Redis-server`) and ramification (`Sensitive.tar` read by process `scp`) of this attack using this subgraph.

As can be seen in Table 4.2, 14% of attack-related vertices (yellow vertices) were spilled to the disk. The main reason for this was our conservative Global List size ($k = 3000$); these attack-related vertices fell outside of the top-k most suspicious paths, leading to their eviction from the suspicious cache. *We found in our experiments that increasing the Global list size from* $k = 3000$ *to* $k = 5000$ *was sufficient to store 100% of attack-related vertices in the cache.* In considering the $k = 3000$ configuration, some temporary files created by the `Redis-server` process, such as `/redis-3.0.3/temp-18434.rdb`, are assigned low suspicious scores because redis regularly creates many such files. However, the temporary file `~/.ssh/temp18434.rdb` was highly unusual because `Redis-server` never writes to the `~/.ssh` folder. As a result, it had a high suspiciousness score and was retained in cache. Note that missing some temporary files from the causal graph does not break causal analysis since we can still identify the root

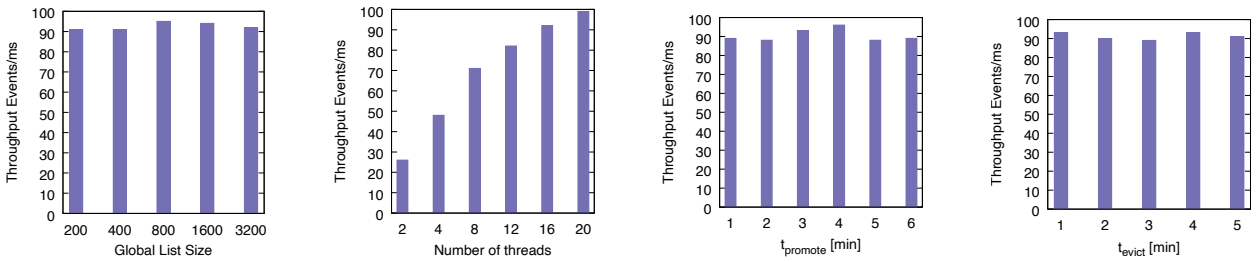


Figure 4.8: Throughput of SWIFT under different configuration values.

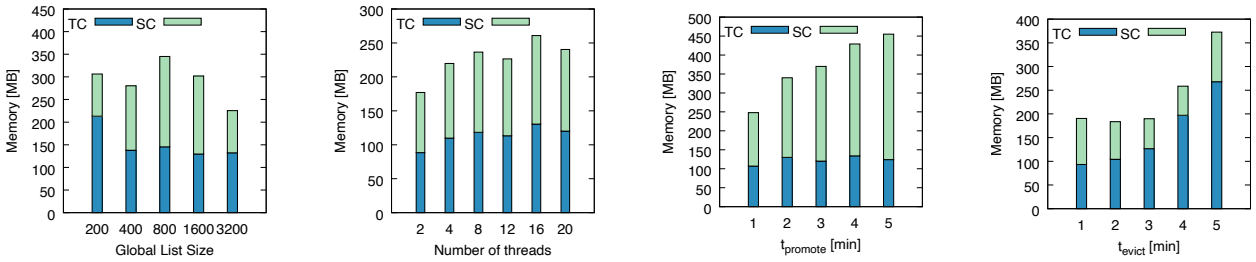


Figure 4.9: Max. memory usage of SWIFT under different configuration values when ran for one day. TC stands for tracking cache and SC stands for suspicious cache.

cause and ramifications using red vertices alone. Further, cyber analysts can still retrieve these yellow vertices from disk later for further investigation.

RQ3: Scalability

Throughput. We define the throughput of SWIFT as the maximum number of events that SWIFT can process under different configuration values of the global list size k, the eviction time window ΔT_{evict}, the promotion epoch $\Delta T_{promote}$, and the number of threads. To stress test SWIFT, we replayed the audit logs from our enterprise engagement at the maximal speed. The results of our throughput experiment are shown in Figure 4.8. Since our eviction algorithm is *asynchronous*, the throughput does not change under different configurations except when we change the number of consumer threads. We can see that SWIFT can process up to 100,000 events/sec when the number of threads is 20, which was the max number of threads allowed by our machine. Note that, in our experiment, each of the 191 hosts generated less than 5,000 events/sec on average, which is far less than the maximal throughput of SWIFT. Assuming that this event generation rate holds, *our prototype implementation can scale to support up to 4,000 hosts with a single server.*

Memory Usage. Another aspect of scalability is memory usage. In our implementation of SWIFT, memory is consumed by two components: the Kafka framework and the cache for events. Since Kafka is only used as a black-box infrastructure in our implementation and could have very different configurations in practice, we focus on the memory usage of the cache. In our experiment, we first measured the maximum memory used by both tracking cache and suspicious cache under different configuration values while monitoring all the 191 hosts for one day. The results are shown in Figure 4.9. Changing global list size and threads does not affect the maximum usage of tracking cache and suspicious cache. Increasing the $\Delta T_{promote}$ increases the size of tracking cache because events stay longer there. On the other hand, increasing ΔT_{evict} window increases the suspicious cache usage since eviction algorithm runs after long time. Our experiment shows that in general, SWIFT could process the workload for 191 hosts in an enterprise with 300 MB memory. For a server with 64 GB memory, as we have used in our experiment, it is possible to handle thousands of hosts at the same time.

RQ4: Benefits of Time Saved

Using causal analysis in state-of-the-art alert triage systems [30], it takes on average 1 min to respond to forensic queries, with a worst case performance of 2.5 hours; because response time grows linearly with graph size, we can expect alerts related to sophisticated intruders to fall closer to this worst-case because they employ a "low and slow" attack approach. On the other hand, SWIFT responds to queries in just 0.1 sec on average, with worst case performance of 1 minute. This effectively provides investigators with alert context (i.e., causal graphs) as soon as the alert is triggered.

Still, it could be argued that an average response time of 1 minute (as opposed to SWIFT's 100 milliseconds) is suitably fast for cyber analysts. However, *it is important to consider the fact investigation latency compounds as the number of alerts increases.* Recent studies [5, 6, 8] have shown that organizations receive around 10,000 alerts per week. For simplicity, let us assume that all 10,000 alerts need to be investigated,[4] and that a true attack falls at each quartile (i.e., alerts 2500, 5000, etc.) of the stack. For the first quartile, NoDoze [30] imposes at least 41 hours of latency due to causal analysis, while SWIFT will impose just 4 minutes of latency. By the last quartile, NoDoze will have imposed 166 hours of latency, while SWIFT introduces just 16 minutes. Further, we can assign a financial cost to

[4]In practice, alert triage systems may be used to condense or procedurally exclude some alerts so that they need not be investigated; however, this exercise demonstrates the value of eliminating causal analysis latency from the threat investigation process.

this difference – studies have shown that it costs an organization $32,000 for every day an attacker stays in the network [100]. Thus, for just the attack in the last quartile, SWIFT could save the organization up to $221,244 as compared to the previous state-of-the-art.

RQ5: Effectiveness in Alert Management

To answer this research question, we measured the performance and accuracy of SWIFT as an alert management system. We used HSM's suspicious influence scores for all the alerts and triaged alerts based on scores. After that we compared our accuracy and performance with the baseline approach [30]. Since our suspicious influence scores were similar to the baseline approach, SWIFT has the same accuracy (false and true positive rates) as the baseline approach. However, the performance of SWIFT is magnitudes of times better than baseline.

The performance of SWIFT over the baseline approach is measured in terms of response time. As we have already shown in the CDF in Figure 4.6 that it took total of 5 hours to rank all the 140 alerts using baseline. The reason for this is that baseline as an offline approach first generate the causal graph using disk storage for each alert. After that, it assigns suspiciousness influence scores to each alert's graph and then triage them based on these scores. On the other hand it took SWIFT around 1 minute to rank all the 140 threat alerts because it generates causal graph in an online fashion and assigns suspiciousness influence scores as events arrive and keeps most suspicious causal graphs in the main-memory. Thus, as soon as alerts are fired by underlying TDS, SWIFT already has its graph with suspiciousness score and just need to lookup this score from the cache to triage which $\mathcal{O}(1)$ time. Since SWIFT also keeps track of all the alerts fired on causal graph, it instantaneously correlates new alert with all the previous alerts that are causally related.

4.5 DISCUSSION & LIMITATIONS

Design of suspicious influence scoring system. SWIFT is effective with an arbitrary suspicious influence scoring system that satisfies all three properties described in Section 4.1.2. In Section 4.4.1, we implemented an anomaly-based scoring system as the reference in our evaluation. Other scoring systems, such as rule-based or label-propagation-based systems, can also be applied as long as they meet the three requirements.

Possible Attacks. One possible attack to spoil the cache of SWIFT is by exploiting Hypothesis H1 – the adversary may conduct an attack in a longer time window so that the causal paths of the attack in the cache are eventually replaced by causal paths of other attacks and

suspicious activities. This attack can be alleviated by allocating large memory (Global List size). As long as there is enough space, SWIFT will maintain the suspicious causal paths in the cache. If there is not enough memory space, choosing which suspicious paths to keep in the cache would be a trade-off. We leave this discussion for our future work. Adversaries may also try to spoil the cache of SWIFT by generating anomalous events and causal paths in provenance data. This can be solved by having more accurate underlying anomaly detection techniques. In this paper, we apply one commercial tool [36] to detect anomalies. Although it is important to improve the accuracy of anomaly detection, it is orthogonal to our study. Nevertheless, even if the cache is spoiled, an investigator can still generate a complete causal graph but with some delay due to disk IO.

Adversaries may try to spoil the cache system of SWIFT to degenerate its responsiveness by having a "spoofing attack". The adversary may conduct an attack that contains a vertex that is involved in more than K most suspicious paths to occupy the whole global list (e.g. unzipping more than K files from a .ZIP package). Under the "spoofing attack," causal paths of other vertices are evicted to the disk so the performance of SWIFT to investigate other vertices is degraded to existing offline solutions. To address this attack, SWIFT only selects candidates from the $PATH_{abnormal}$ of each vertex. Since the size of $PATH_{abnormal}$ of each vertex is limited to m, each vertex will only occupy at most m slots in the global list.

Another type of spoofing attack is that the adversary may generate a lot of different suspicious events to occupy the cache. Since we keep the longest path in the cache, the adversary needs to generate a huge number of **independent** suspicious events, which do not have causal dependencies, to spoil the cache. However, if an attacker tries to produce a lot independent suspicious events then it defeats the "low and slow" strategy used by attackers and generates a strong indication of an attack which a threat hunter can immediately spot. Moreover, this problem is equivalent to the problem of having too many suspicious paths that the cache cannot hold, which we leave for future work.

Applicability. The SWIFT approach is generic to provide broad support for fast and interactive threat hunting in enterprises provided that system-level audit logs are being collected and there is an underlying threat detector which monitors enterprise-wide activities. The two key hypotheses presented in Section 4.3, upon which SWIFT is built, are enterprise agnostic. These hypotheses are derived from fundamental characteristics of system-level audit logs [55, 56]. This ensures that our techniques can be applied in different enterprises without sacrificing performance and accuracy.

CHAPTER 5: OMEGALOG: GENERATING ACCURATE AND SEMANTICS-AWARE PROVENANCE GRAPHS

Given the importance of threat investigation to system defense, it is perhaps surprising that prior work on causality analysis has been oblivious to application-layer semantics. As an example, consider the execution of the web service shown in Figure 5.1. Figure 5.1(a) describes the event sequence of the example, in which the server responds to two HTTP requests for `index.html` and `form.html`, respectively, yielding the system log shown in Figure 5.1(b). As a normal part of its execution, the server also maintains its own event logs that contain additional information (e.g., user-agent strings) shown in Figure 5.1(c), that is opaque to the system layer. State-of-the-art causality analysis engines, using system audit logs, produce a provenance graph similar to Figure 5.1(d); however, the forensic evidence disclosed by the application itself is not encoded in this graph. That is unfortunate, as recent studies [123, 124, 125] have shown that developers explicitly disclose the occurrence of *important* events through application logging. Further, we observe that the well-studied problem of *dependency explosion* [13, 16, 21], which considers the difficulty of tracing dependencies through high-fanout processes, is itself a result of unknown application semantics. For example, the dependency graph in Figure 5.1 (d) is not aware that the NGINX vertex can be subdivided into two autonomous units of work, marked by the two HTTP requests found in the application event log.

Prior work on log analysis has not provided a generic and reliable (i.e., causality-based) solution to cross-layer attack investigation. Techniques for execution partitioning mitigate dependency explosion by identifying limited and coarse-grained application states, e.g., when a program starts its main event-handling loop [13], but require invasive instrumentation [13, 16] or error-prone training [13, 14, 21]. Past frameworks for layered provenance tracking [10, 126, 127, 128] technically support application semantics, but rather than harness the developer's original event logs, instead call for costly (and redundant!) instrumentation efforts.

Elsewhere in the literature, application event logs have been leveraged for program debugging [129, 130, 131], profiling [132, 133], and runtime monitoring [134]; however, these approaches are application-centric, considering only one application's siloed event logs at a time, and thus cannot reconstruct complex workflows between multiple processes. Attempts to "stitch" application logs together to trace multi-application workflows [88, 132, 133] commonly ignore the system layer, but also use ad hoc rules and *co-occurrence* of log events to assume a *causal* relationship; this assumption introduces error and could potentially undermine threat investigations.

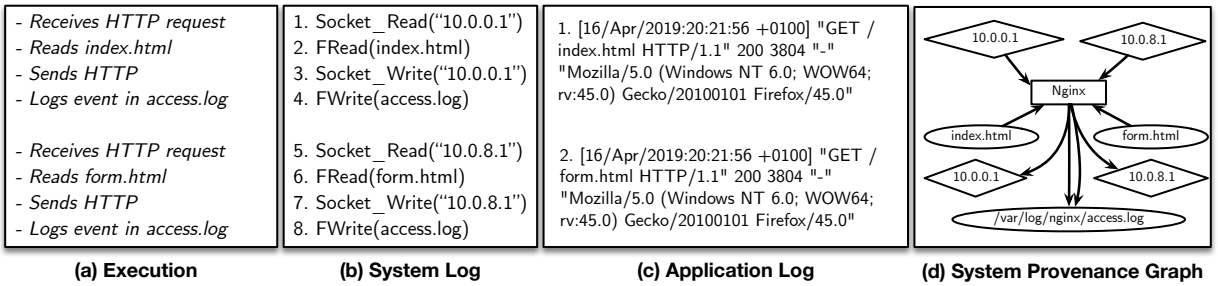


Figure 5.1: NGINX application execution while two different HTTP requests are being served. **(a)** Actual execution behavior of NGINX. **(b)** System logs generated by whole-system provenance tracker. **(c)** Application event logs generated by NGINX. **(d)** Provenance graph generated using system logs by traditional solutions.

In this chapter, we argue that attack investigation capabilities can be dramatically improved through the unification of *all* forensically relevant events on the system in a single holistic log. To achieve that vision transparently and effortlessly on today's commodity systems, we present OmegaLog, an end-to-end provenance tracker that merges application event logs with the system log to generate a *universal provenance graph* (UPG). This graph combines the causal reasoning strengths of whole-system logging with the rich semantic context of application event logs. To construct the UPG, OmegaLog automatically parses dispersed, intertwined, and heterogeneous application event log messages at runtime and associates each record with the appropriate abstractions in the whole-system provenance graph. Generating UPG allows OmegaLog to transparently solve both the dependency explosion problem (by identifying event-handling loops through the application event sequences) and the semantic gap problem (by grafting application event logs onto the whole-system provenance graph). Most excitingly, OmegaLog does not require any instrumentation on the applications or underlying system.

Several challenges exist in the design of a universal provenance collection system. First, the ecosystem of software logging frameworks is heterogeneous, and event logging is fundamentally similar to any other file I/O, making it difficult to automatically identify application logging activity. Second, event logs are regularly multiplexed across multiple threads in an application, making it difficult to differentiate concurrent units of work. Finally, each unit of work in an application will generate log events whose occurrence and ordering vary based on the dynamic control flow, requiring a deep understanding of the application's logging behavior to identify meaningful boundaries for execution unit partitioning.

To solve those challenges, OmegaLog performs static analysis on application binaries to automatically identify log message writing procedures, using symbolic execution and emulation to extract descriptive Log Message Strings (LMS) for each of the call sites. Then,

OmegaLog performs control flow analysis on the binary to identify the temporal relationships between LMSes, generating a set of all valid LMS control flow paths that may occur during execution. At runtime, OmegaLog then uses a kernel module that intercepts write syscall and catches all log events emitted by the application, associating each event with the correct PID/TID and timestamp to detangle concurrent logging activity. Finally, those augmented application event logs are merged with system-level logs into a unified universal provenance log. Upon attack investigation, OmegaLog is able to use the LMS control flow paths to parse the flattened stream of application events in the universal log, partition them into execution units, and finally add them as vertices within the whole-system provenance graph in *causally correct* manner.

5.1 MOTIVATING EXAMPLE

In this section, we explain the motivation for our approach by considering a data exfiltration and defacement attack on an online shopping website. We use this example to illustrate the limitations of existing provenance tracking systems [10, 12, 15, 16, 21, 43, 108]. Consider a simple WordPress website hosted on a web server. Requests to the website are first received by an HAProxy, which balances load across different Apache instances running on the web server, while customer transactions are recorded in a PostgreSQL database. The administrator has turned on application event logging for Apache httpd, HAProxy, and PostgreSQL. In addition, the server is performing system-level logging, e.g., through Linux Audit (auditd) [56] or Linux Provenance Modules (LPM) [10], which continuously collect system logs. One day, the administrator discovers that the online store has been defaced and that some of the sensitive customer information has been posted to a public Pastebin website. On average, the shopping website receives tens of thousands of requests per day; among those, one request was malicious.

5.1.1 Investigating with Application Event Logs

To attribute the attack and prepare an appropriate response, the administrator initiates a forensic inquiry by first inspecting the application event logs. The administrator finds that the `accounts` database table must have been accessed and uses this as a *symptom* to initiate attack investigations. The admin then runs a `grep` query on PostgreSQL event logs, which returns the following query log message:

```
SELECT * FROM users WHERE user_id=123 UNION SELECT password FROM accounts;
```

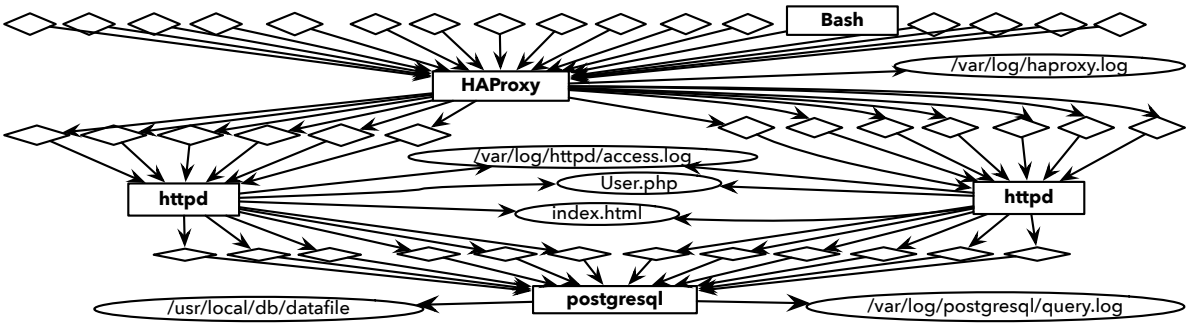


Figure 5.2: A whole-system provenance graph showing the SQL injection attack scenario. Diamond, box, and oval vertices represent network connections, processes, and files, respectively. This graph suffers from both dependency explosion and semantic gap problems, frustrating attack investigation.

This log message strongly indicates that an attacker exploited a SQL injection vulnerability in the website, and also suggests that the attacker was able to retrieve the login credentials for `admin.php` which gave attacker privileged site access.

Limitations of Application Event Logs. At this point, the administrator is unable to proceed in the investigation using application event logs alone. It is clear that the HAProxy and Apache httpd logs contain important evidence such as the HTTP requests associated with the SQL injection attack, but re-running of the same `grep` query on Apache's logs did not return any result. The reason is that the attacker used a `POST` command to send the SQL query and that command was not contained in the URL captured in the Apache httpd event log messages. The investigation has stalled with important questions left unanswered: 1) *What was the IP address associated with the malicious HTTP request?* 2) *How were the login credentials used to deface the website, and what additional damage was caused?* 3) *Which PHP file on the site is not properly sanitizing user inputs, exposing the SQL injection vulnerability?* Those questions reflect an inherent limitation of application event logs: they cannot causally relate events across applications and thus cannot trace workflow dependencies.

5.1.2 Investigating with System Logs

To proceed, the administrator attempts to perform causality analysis using a whole-system provenance graph. At this layer, it is easy to trace dependencies across multiple coordinated processes in a workflow. Because the malicious query shown above resulted in a read to the PostgreSQL database, the administrator uses `/usr/local/db/datafile.db` as a *symptom* event and issues a backtrace query, yielding the provenance graph shown in Figure 5.2.

Table 5.1: Comparison of execution partitioning techniques to solve the dependency explosion problem.

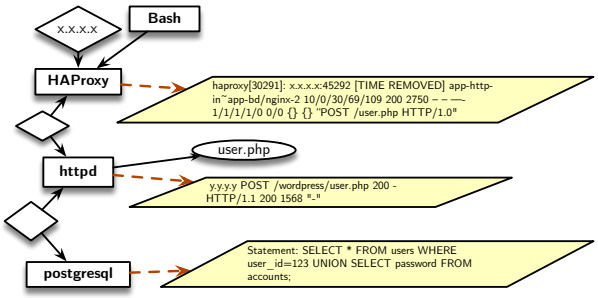
	BEEP [13] ProTracer [21]	MPI [16]	MCI [12]	WinLog [14]	**OmegaLog**
Instrumentation	Yes	Yes	No	No	**No**
Training Run w/ Workloads	Yes	No	Yes	No	**No**
Space Overhead	Yes	Yes	Yes	Yes	**No**
Granularity	Coarse	Fine	Coarse	Coarse	**Fine**
App. Semantics	No	No	No	No	**Yes**

Unfortunately, the administrator discovers that this technique does not advance the investigation because of the inherent limitations of system logs.

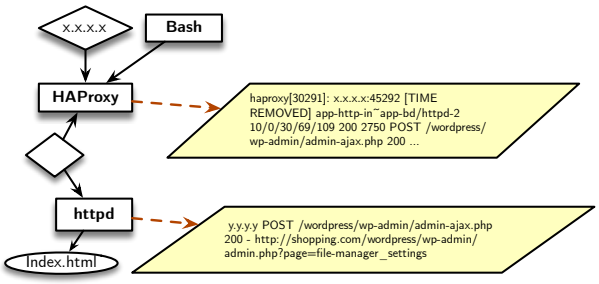
Limitation of System Logs #1: Dependency Explosion. The administrator's backtrace identifies thousands of "root causes" for the SQL injection attack because of the dependency explosion problem. The reason is that system-layer provenance trackers must conservatively assume that the output of a process is causally dependent on *all* preceding process inputs [12, 13, 16, 21]. Although the malicious query string is known, causal analysis does not allow the administrator to associate the query with a particular outbound edge of `/usr/local/db/datafile.db` in the provenance graph. Even if the administrator restricted most of the dependencies between Apache httpd and PostgreSQL (e.g., though timing bounds), admin would again face the same problem when identifying which input request from HAProxy to Apache httpd lies on the attack path.

Recent work [13, 14, 21] has introduced *execution partitioning* as a viable solution to the dependency explosion problem. These systems decompose long-running processes into autonomous "units", each representing an iteration of event-handling loop, such that input-output dependencies are traced only through their corresponding unit. Where event handling loops do not encode work units, Kwon *et al.* propose an inference-based technique for identifying units from system log traces [12] while Ma *et al.* propose a framework for manually annotating source code to disclose meaningful unit boundaries [16].

Unfortunately, prior approaches suffer from noteworthy limitations, which we summarize in Table 5.1. Most execution partitioning systems rely on *instrumentation* to identify unit boundaries, requiring either domain knowledge or manual effort and assuming the right to modify program binaries, which is not always available [14]. The common requirement of *training runs* exposes systems like BEEP and Protracer to the classic code-coverage problem present in any dynamic analysis, and inference-based techniques (MCI) may also struggle with out-of-order events due to the presence of concurrent or cooperating applications during training runs. All past approaches introduce additional *space overhead* in order to track unit



(a) Investigating SQL injection attack using SQL query that reads the `accounts` table.



(b) Investigating website defacement using a file write event to `index.html` as a symptom.

Figure 5.3: Graphs generated by OmegaLog for the SQL injection attack. The parallelograms represent the *app log vertices*. App log vertex is annotated with log messages which belong to the corresponding execution unit of attached process vertex.

boundaries; fully automated identification of event loops (BEEP, Protracer) can generate excessive units that can waste space and CPU cycles [16]. Most notably, prior approaches do not consider the broader value of *application semantics* as forensic evidence outside of the bare minimum required for the identification of work units.

Limitation of System Logs #2: Semantic Gap. Existing system-level provenance logs are beneficial in that they offer a broad view of system activity, but unfortunately they lack knowledge of application-specific behaviors that are pivotal for attack reconstruction. In our motivating example, information such as failed login attempts, HTTP headers, WordPress plugin behavior, and SQL queries cannot be extracted from system logs. Such information is present in the siloed event logs of each application; PostgreSQL maintained a record of all SQL queries, and HAProxy recorded the headers for all HTTP requests. However, it is not possible to associate those event descriptions with the system records reliably in a post-hoc manner, because of multi-threaded activity and ambiguous or incomplete information within the application event logs.

Prior work has sought to address the semantic gap problem through instrumentation-based techniques [126, 127, 135]. Those approaches either statically or dynamically instrument function calls in the application to *disclose* function names, arguments, and return values. However, such instrumentation-based systems suffer from several limitations: (1) developers need to specify which functions to instrument, imposing a domain knowledge requirement; (2) the logging information is captured on a per-application basis and thus cannot be used to connect information flow between different applications; and (3) high-level semantic events may not always be effectively captured at the function call level.

5.1.3 OmegaLog Approach

Recent work in application logging [123, 124, 125, 132, 133] has shown the efficacy of application logs in program understanding, debugging, and profiling. OmegaLog takes inspiration from those efforts, with the goal of better leveraging event logs during attack investigation. *The key insight behind OmegaLog is that developers have already done the hard work of encoding high-level application semantics in the form of event logging statements*; these statements not only contain the relevant forensic information that we require, but also mark the boundaries of execution units in the program. The insertion of event logging statements is an organic byproduct of sound software engineering practices, permitting developers and users to better understand programs' runtime behavior. Thus, it is possible to enrich system logs with application semantics without further instrumentation or profiling. Moreover, these applications logs can be used to identify execution units.

Applying that intuition to our motivating example yields the provenance graph in Figure 5.3a, which was generated using OmegaLog. The administrator can associate the malicious SQL query with a specific system call event (`read`). By performing execution partitioning on PostgreSQL using OmegaLog's logging behavior analysis, the administrator is then able to trace back to system calls issued and received by Apache httpd, which are also annotated with application events describing the vulnerable web form. Iteratively, OmegaLog uses execution partitioning again to trace back to the correct unit of work within HAProxy to identify the IP address of the attacker. After finding out how the user data and login credentials were stolen using SQL injection, the investigator tries to figure out how the website was defaced by issuing a backward-tracing query on the `index.html` file. Using the OmegaLog provenance graph shown in Figure 5.3b, the investigator deduces that the attacker used a WordPress file manager plugin to change `index.html`.

5.2 BACKGROUND: APPLICATION LOGGING BEHAVIOUR

Our approach to partition long-running program into execution units and overcome the dependence explosion problem depends on the pervasiveness of event-logging behavior in those applications. Fortunately, the importance of logging in applications has been widely established [136]. Practically, all open-source applications print event log messages, offering four levels of verbosity: *`FATAL`* is for an error that is forcing a shutdown, *`ERROR`* is for any error that is fatal to the operation, *`INFO`* is for generally useful information, and *`DEBUG`* is for information that is diagnostically helpful. Note that logging levels are inclusive; higher levels also print messages that belong to lower levels (*i.e.*, `FATAL` $\subseteq$ `ERROR` $\subseteq$ `INFO` $\subseteq$ `DEBUG`).

Table 5.2: Logging behavior of long-running applications.

Category		Total Apps	Apps with Log Verbosity of			
			IN+DE	INFO	DEBUG	None
Client-Server	Web server	9	7	1	0	1
	Database server	9	7	1	1	0
	SSH server	5	5	0	0	0
	FTP server	5	4	0	1	0
	Mail server	4	3	1	0	0
	Proxy server	4	3	1	0	0
	DNS server	3	2	0	1	0
	Version control server	2	0	1	1	0
	Message broker	3	2	0	1	0
	Print server	2	1	0	1	0
	FTP client	6	0	1	4	1
	Email client	3	1	0	1	1
	Bittorrent client	4	3	1	0	0
	NTP client	3	0	1	2	0
GUI	Audio/Video player	8	1	0	3	4
	PDF reader	4	0	0	0	4
	Image tool	5	0	0	1	4
Total		**79**	**39**	**8**	**17**	**15**

However, to partition successful executions of an application into its units, we require log messages with verbosity level of `INFO` or `DEBUG` to be present *inside* event-handling loops. Unfortunately, such behavior in applications has not been investigated. In that regard, we studied a large number of popular open-source applications.

We collected a list of 79 long-running Linux applications which belong to different categories. Those applications are written in the C/C++, Java, Python, and Erlang programming languages. We investigated the source code and man pages of those applications to identify the event-handling loops and understand if they print log messages for each meaningful event. Lee et al. [13] conducted a similar study in 2013 but they only analyzed the design patterns of open-source applications and the pervasiveness of event-handling loops as drivers for execution. They did not however study the logging behavior of those applications and the presence of log messages inside event-handling loops.

We summarize our results in Table 5.2. In the column "Apps with Log Verbosity of", we show how many of 79 profiled applications include log statements in their event-handling loop at verbosity of `INFO` and `DEBUG`, and how many of 79 applications do not print meaningful log messages for new events. We observe that 39 applications print log with both `INFO` and `DEBUG` verbosity levels (`IN+DE`) inside the event-handling loops. While 8 applications only log at `INFO` level and 17 applications only log at `DEBUG` level. For web servers such as lighttpd and NGINX, we treat the Access Log as INFO level log. Moreover, for certain applications

that do not have `DEBUG` log level, we categorize the Trace Log as `DEBUG` level log. We show the intra-event-handling loop logging behavior of some of the well-know applications in Figure 5.4.

During our study, we found 15 applications that do not have any information about event logs in their source code or in man pages. We categorized those applications as follows:

- **Light-weight Applications:** Certain client-server applications are designed to be light-weight to keep a minimal resource footprint. Those applications – including `thttpd` (Web server) and `skod` (FTP client) – do not print log messages for new events.
- **GUI Applications:** We observe that 12 out of 17 GUI applications either (1) do not print log messages, or (2) they print log messages that do not match the expectations of the forensic investigator. In other words, those log messages were not meaningful to partition the execution. Ma et al. [16] also observed similar behavior for GUI applications where event-handling loops do not correspond to the high-level logic tasks. For example, we found that none of the PDF readers in our study printed log messages whenever a new PDF file was opened. Such PDF file open event is forensically important event for threat investigations [16].

Our study suggests that sufficient logging information is present inside the event-handling loops of long-running applications. This behavior allows us to automatically identify the unit boundaries of those programs. For further evaluation, we only consider the applications shown in Table 5.3. We picked those applications based on their popularity and category. Note that we did not pick any subjects from the category of applications that do not print meaningful log messages for new events. Moreover, GUI applications usually use asynchronous I/O with call backs and such programming model is not currently handled by OmegaLog (described more in Section 5.8).

5.3 DESIGN OVERVIEW

5.3.1 Properties of Causality Analysis

The provenance graph generated by OmegaLog should preserve the following three properties of causality analysis.

- *Validity* means that the provenance graph describes the correct execution of the system, i.e., the provenance graph does not add an edge between entities that are not causally related.

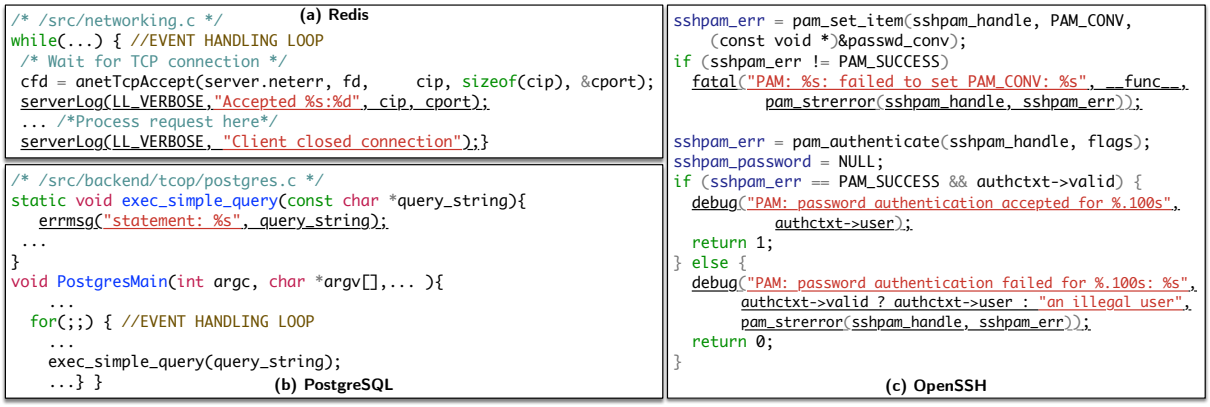


Figure 5.4: Logging behavior of different applications inside the event-handling loop. Underlined code represent log printing statements.

- *Soundness* means that the provenance graph respects the happens-before relationship during backward and forward tracing queries.
- *Completeness* means that the provenance graph is self-contained and fully explains the relevant event.

5.3.2 Design Goals

The limitations mentioned in Section 5.1 on prior work motivated our identification of the following high-level goals for OmegaLog:

- **Semantics-Aware.** Our threat investigation solution must be cognizant of the high-level semantic events that occurred within the contexts of each attack-related application.
- **Widely Applicable.** Our solution must be immediately deployable on a broad set of applications commonly found in enterprise environments. Therefore, the solution must not depend on instrumentation or developer annotations. Moreover, our techniques should be agnostic to applications' system architecture and should apply to proprietary software, for which source code is usually not available.
- **Forensically Correct.** Any modifications made to the whole-system provenance graph by our solution must support existing causal analysis queries and preserve the properties of validity, soundness, and completeness.

5.3.3 OmegaLog

Fig. 5.5 presents a high-level overview of the OmegaLog system, which requires that both system-level logging and application event logging be enabled. OmegaLog's functionality is

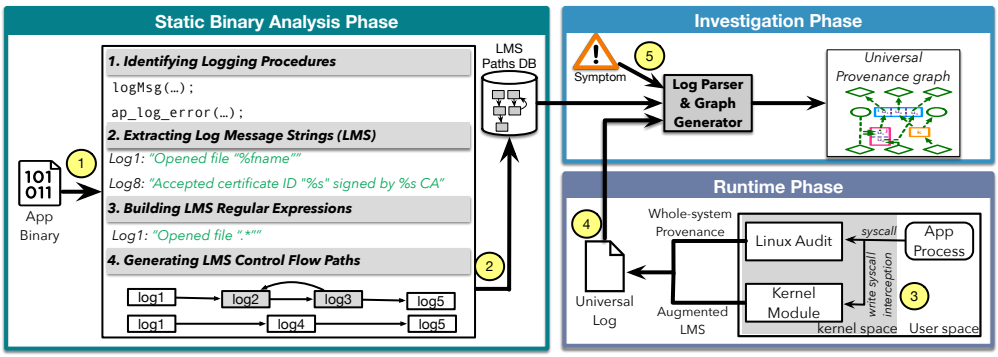


Figure 5.5: OmegaLog architecture overview. During the offline phase, OmegaLog first generates control flow graph and extracts log message strings (LMSes) from application's binary and then contructs LMS control flow paths. During the runtime phase, OmegaLog combines application event logs and audit logs together into universal provenance logs. Finally, during the investigation phase, OmegaLog uses LMS control flow paths to parse universal provenance log into universal provenance graphs.

divided into three phases: static binary analysis (Section 5.4), runtime (Section 5.5), and investigation (Section 5.6). In the static analysis phase, (①) OmegaLog first analyzes all application binaries to extract all log message strings (LMSes) that describe event-logging statements in the code, and then uses control flow analysis to identify all possible temporal paths of LMS in different executions of the program. (②) All those LMS control flow paths are stored in a database that is input to a log parser to bootstrap interpretation of application events. At runtime, (③) OmegaLog captures all the application events and augments them with the application's PID/TID and a timestamp of log event through kernel module that intercepts write syscalls. Simultaneously, (④) OmegaLog collects system logs from the underlying whole-system provenance tracker and associates them with the appropriate application events by using the PID/TID as a disambiguator; and store them into a unified log. Upon attack investigation, (⑤) OmegaLog passes that universal log and the LMS control flow paths database to a log parser that partitions associated processes in the whole-system graph by inserting a new *app log vertex*. This vertex is connected to the corresponding partitioned process and annotated with log messages in that particular execution unit of the process. The semantic-aware and execution-partitioned graph is called universal provenance graph (UPG), which is presented to the investigator.

5.4 STATIC BINARY ANALYSIS PHASE

The static analysis routine profiles application binaries before their execution. During static analysis, OmegaLog performs several passes over the binary's *control flow graph* (CFG)

to identify logging behaviors and generate all possible LMS paths that are possible during execution of that binary. Specifically, we leverage the Angr [137] toolchain to build the CFG, and then introduce new methods to automatically identify logging procedures in the binary (§5.4.1). Next, we concretize LMS (§5.4.2) using the identified logging procedure, and finally we generate all possible LMS control flow paths that can occur during execution of the binary (§5.4.4). Those steps are also shown in Fig. 5.5.

As highlighted in earlier work [138], binary analysis imposes high costs, especially when symbolic execution and emulation are necessary. In what follows, we describe how OmegaLog avoids prohibitive analysis costs while profiling application-logging behaviors. Although, OmegaLog works on application binaries, for convenience, we explain static analysis procedures by using source code snippets. Algorithm 5.1 offers a high-level overview of our static analysis routines.

5.4.1 Identifying Logging Procedures

The ecosystem of event-logging frameworks is diverse and heterogeneous; to overcome the resulting issues, OmegaLog identifies logging procedures in a binary by using two heuristics. 1) Applications use either well-known libraries (e.g., `syslog` [139], `log4c` [140]) or functionally-similar custom routines to produce, store, and flush log messages to a log file. The libraries leverage the I/O procedures of `Libc`, such as `fprintf` or `snprintf`, to write the log messages to disk. OmegaLog can thus identify candidate logging procedures through a backward traversal of the CFG from these procedures call sites. 2) Most applications that create event logs store messages in the `/var/log/` directory by default. Thus, OmegaLog can differentiate log I/O from other I/O based on the file path and consider all the procedures that write to `/var/log/` directory as logging procedures. Combining these two heuristics was sufficient to identify logging behaviors for applications in our evaluation dataset. Nevertheless, OmegaLog also provides an interface that sysadmins can use to add the names of their logging procedures, if the binary does not follow the aforementioned conventions.

5.4.2 Extracting Log Message Strings (LMS)

Once we have identified all the logging procedure names in the previous step, we assign a unique identifier for each logging procedure callsite. We need to generate an LMS that describes the format specifier arguments (template) of the log message. This step requires OmegaLog to extract the binary's full control flow graph and perform symbolic execution [141] to extract the values of such arguments. We henceforth refer to this process

Algorithm 5.1: Static Binary Analysis

```
1  Func GetLMS(Binary B, Log functions F)
       /* Overall process to build the LMS paths                          */
2      g ← AngrGetFastCFG(B)
3      C ← ExtractCallSites(g, F)
       /* Concretization step                                             */
4      V ← PeepholeConcretization(g, C)
       /* Building the LMS paths step                                     */
5      G ← BuildLMSPaths(g, V, F)

6  Func ExtractCallSites(cfg, F)
7      C ← Φ
8      foreach basic block b ∈ cfg do
           /* Check if the basic block jumps into a logging function      */
9          if b.jump_target_address ∈ F.addresses then
10             C ← C ∪ {b}
11         end
12     end
13     return C

14 Func PeepholeConcretization(cfg, call_sites, maxBackTrace)
15     V ← Φ
16     V ← {(b, 0) for b ∈ call_sites}
17     while V ≠ Φ do
18         (b, backtrace) ← V.pop()
           /* L is of the form {(LMS ℓ, call_stack cs)}                   */
19         L ← SymbolicExecution(g, v)
20         if L ≠ Φ then
21             foreach (ℓ, cs) ∈ L do
                   /* Taking care of context sensitivity                  */
22                 topBlock ← cs.top()
23                 if (ℓ, topBlock) ∉ V then
24                     V ← V ∪ {(ℓ, topBlock)}
25                 end
26             end
27         end
28         else if backtrace ≤ maxBackTrace then
29             V ← V ∪ {(v, backtrace + 1) for v ∈ b.predecessors()}
30         end
31     end
32     return V

33 Func BuildLMSPaths(cfg, V, F)
       /* E is the set of paths between LMS                               */
34     E ← Φ
35     foreach f ∈ cfg.functions()\{F} do
           /* Extract the entry points and external returns               */
36         entries ← f.entry_points()
37         returns ← f.jumps()
38         E ← E ∪ GetLocalPaths(V, f)
39     end
```

as *concretization*. However, performing a complete symbolic execution over the binary is a computationally expensive operation that leads to the path explosion problem, especially for applications with complex compile-time optimizations. In fact, while experimenting with the applications listed in Table 5.3, we realized that most applications are compiled with at least the `-O2` compiler optimization level, which greatly complicated the task of CFG extraction

and symbolic execution. For example, when we used the Angr toolset, extracting the CFG and performing symbolic execution on the `openssh` server binary quickly exhausted 64 GB of memory on our experimental machine and did not return a conclusive result, even after running for several hours.

To overcome that problem, we first note that our exclusive purpose is to obtain the format specifier arguments for logging function calls; any symbolic execution operation that does not serve this purpose is unnecessary. Therefore, OmegaLog first references the CFG built without symbolic execution (referred to as a `FastCFG` in Angr toolset), which is generated by traversing the binary and using several heuristics to resolve indirect jumps; that approach greatly reduces the CFG computational and memory requirements [137]. Using the `FastCFG`, we identify the basic blocks that contain function calls or jumps to logging procedures, and thus we can focus our attention solely on such blocks. Nevertheless, unlike the full CFG, the `FastCFG` does not retain any state about the binary that would allow OmegaLog to concretize the values of the logging procedures' arguments.

To complete our analysis, we introduce an optimized concretization we refer to as *peephole concretization*. While studying the code of the open-source programs shown in Table 5.3, we observed that for the most part, format specifier arguments to logging procedures are passed either (1) as direct constant strings or (2) through constant variables defined near the procedure call. For example, consider the call to the `debug` logging procedure in the `OpenSSH` application shown in Fig. 5.4. The LMS we are interested in extracting is the message `‘‘PAM: password authentication accepted for %.100s’’` passed directly as a constant to the function call. At the machine instructions level, that observation reflects the fact that LMSes are typically defined within the same basic block that ends with the `call` or `jump` instruction to the address of a logging function, or in a nearby preceding block.

Using peephole concretization, we only need to perform local symbolic execution starting from the basic blocks identified in the previous step, stopping directly after executing the call instruction to the target logging procedure. We show the pseudocode for our peephole concretization step in Algorithm 5.1. If the symbolic execution task of a given basic block b fails to concretize LMS values, OmegaLog then launches new symbolic execution tasks from each of b's predecessors (referred to as $b.predecessors()$ in Algorithm 5.1). We refer to the operation of restarting symbolic execution from a basic block's predecessors as *backtracing*. OmegaLog bounds the computational resources employed for the concretization step by halting symbolic execution after performing `maxBackTrace` backtrace operations from a given block b. If symbolic execution fails to produce concretized LMS values after `maxBackTrace` operations, OmegaLog marks the function as *unresolved* and produces incomplete LMS paths.

Our algorithm may yield ambiguous LMS paths in the rare cases in which the function call can have different format specifiers based on the sequence of basic blocks that lead to it (i.e., *context sensitivity*). We address that challenge during the peephole concretization step by recording the call stack that produced each LMS. If two different call stacks produce different LMS for the logging function call, we create a new LMS for each call and then associate it with the topmost basic block on each corresponding function call. That process will guarantee that we do not miss any LMS and that we do not over-approximate the reachability between LMSes when constructing the LMS control flow paths. We note, however, that making format specifiers to logging procedures context-dependent is not a frequently observed programming practice; in fact, we encountered this issue only when processing the `transmission` and `CUPSD` applications.

5.4.3 Building LMS Regular Expressions

Finally, once an LMS has been concretized, we can extract a regex that can be used to match event messages at runtime. The resulting regex describes the format specifiers in the LMS that depend on runtime context (e.g., `%s`, `%d`, `%%s`). Each format specifier is replaced with a suitable regex, e.g., "`%d`" with "`[0-9]+`" and "`%s`" with "`.`". For example, one LMS we encounter in `OpenSSH` is

```
PAM: password from user %.12s accepted.
```

After extraction, that yields the following regex:

```
PAM: password from user .* accepted.
```

5.4.4 Generating LMS Control Flow Paths

After concretizing LMS with selective symbolic execution, OmegaLog can continue to use the `FastCFG` to enumerate the valid sequences of LMS that can appear in a typical lifecycle of the application. Extraction of all the possible paths is not a direct application of depth-first traversal (DFS); DFS renders an under-approximation of the possible paths for the following reasons. (1) The same basic blocks can be called from different callees and thus must be traversed multiple times. (2) Function calls (i.e., `call` instructions) must be matched with their appropriate `return` or `jump` instructions. Finally, (3) the applications we study use an abundance of loops and recursive functions that must be traversed multiple times in order to

avoid skipping over loop paths. Instead, our approach addresses (1) and (2) by using caching and temporary nodes, and (3) by using *fixed-point* iterations. Pseudo-code for OmegaLog's control flow path building algorithm (BuildLMSPaths) is given in Algorithm 5.1.

Instead of traversing the full binary's CFG, OmegaLog subdivides the path identification task into several function-local traversals that generate subgraphs for each function in the binary. It then links these subgraphs by following `call` and `return`/`jump` instructions to build the full LMS paths. For each function f in the binary's functions (referred to as $cfg.functions()$ in Algorithm 5.1), OmegaLog identifies f's entry points, in which control flow passes into the function, and its exit points, in which control flow crosses the f's local body, creating dummy LMS nodes for these points. Then, OmegaLog performs a local traversal of f's subgraph; starting from f's entry points, we traverse the control flow edges between the basic blocks that do not leave f's address space.

Every time OmegaLog encounters a basic block containing an LMS, that block is added to the path, and its outgoing edges are traversed. To accurately capture looping behavior, we perform a *fixed-point* iteration over the loop edges until no further changes occur to the LMS path being built. In other words, we keep traversing the same loop edge until no further LMS paths are detected; we then consider the loop edge to be exhausted and move to the next control flow edge. Finally, to speed up the traversal, OmegaLog caches processed basic blocks so that it needs to only traverse them once if multiple paths coincide. Note that we do not consider any loops that do not contain any syscalls because such loops do not produce audit logs and thus cannot be used for execution partitioning.

After building the function-local subgraphs, OmegaLog resolves the `call` and `jump` instructions in each of them to complete the full LMS paths. For each function call that is on an LMS path, OmegaLog injects the callee's subgraph into the path by creating links between the caller's basic block and the callee's entry points and between the callee's exit points (`return` blocks and `jump` instructions targeting the caller) and the callee's return basic block. Using that approach, OmegaLog completes the full LMS paths while also handling recursive functions by creating self-cycles. Subsequently, OmegaLog compresses the graph by removing the dummy nodes created by the BuildLMSPaths function and merging their fan-in and fan-out edges. The resulting compressed graph will then contain all the detected LMS paths. Fig. 5.6 shows an example of LMS control flow paths from a code snippet. The code is shown on the left, and the corresponding LMS paths are shown on the right. The backedge from `log3` to `log2` just shows that these logs are inside a loop and can appear more than one time.

LMS control flow paths guide OmegaLog to partition universal provenance log into execution units; however, in some applications printed LMSes in the event-handling loop are

```
log("Server started"); // log1
while(...) {
 log("Accepted Connection"); // log2
 ... /*Handle request here*/
 log("Closed Connection"); // log3
}
log("Server stopped"); // log4
```

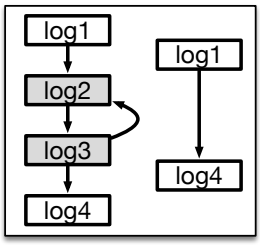


Figure 5.6: On the right, LMS control flow paths representation is shown for the code snippet on the left.

not precise enough to partition the loop. For example, Redis event-handling loop shown in Figure 5.4 prints two LMSes in each iteration of the event-handling loop. The first LMS is printed after the `accept` syscall and if we partition the event-handling loop based on the *both* first and second LMSes, then we will miss that `accept` syscall in the execution unit and only capture syscalls that happened in between two LMSes. However, if we partition the event-handling loop only on the second LMS then we will generate correct execution units because there is no syscall after second LMS in the event-handling loop.

Thus, during LMS control flow paths construction OmegaLog marks all the LMSes present inside the loops that do not have any syscalls before or after in that loop. Such marking helps OmegaLog to make correct execution partitioning of universal provenance log during investigation phase. If there is no such LMS inside the loop then OmegaLog keeps track of either all the syscalls present after the last LMS (loop-ending LMS) in the loop or all the syscalls present before the first LMS (loop-starting LMS) in the loop whichever has least number of syscalls. OmegaLog uses such syscall mappings during investigation phase to make correct execution units.

5.4.5 Discussion of Static Analysis Limitations

Our approach is agnostic to the underlying binary analysis tool, but in this work, we used Angr tool, which came with its own set of limitations. Below we discuss these limitations and, in some cases, how we handled them to recover LMS paths.

False Positives & False Negatives. For more information on accuracy and completeness of Angr's recovered CFG, we refer the reader to [137]. In brief, if Angr mistakenly adds an edge that should not be in the CFG of an application, OmegaLog will generate an erroneous LMS path in the LMS path database. However, since that execution path will never happen during runtime, OmegaLog will just ignore this false positive LMS path during UPG construction. In case Angr misses an edge in a CFG, we have implemented Lookahead and Lookback matching (described in Section 5.6), which handle this case.

Runtime Performance. OmegaLog's static analysis runtime performance was significantly impacted by Angr's performance of symbolic execution. We introduced PeepholeConcretization to improve runtime while preserving the accuracy of LMS path recovery. Note that static analysis is a one-time, offline cost: once a binary has been profiled, there is no need to re-analyze it unless it has been changed. On modestly provisioned workstations, that task could even be outsourced to more powerful machines.

Binary Restrictions. First, Angr tool can only work on binaries compiled from C/C++ code. Second, the format modifier argument to a logging procedure should not be built dynamically at runtime as an element of a `struct`, i.e., it should be a constant string. Third, our binary analysis can only recover logging functions that are not inlined. However, we did not encounter inlined logging functions during our evaluation.

5.5 RUNTIME PHASE

At runtime, OmegaLog performs minimal maintenance of application and whole-system logs; the LMS control flow path models are stored in a database (② in Fig. 5.5) and are not consulted until an investigation is initiated. The primary runtime challenge for OmegaLog is that of reconciling logs from different layers, which is difficult when considering a flattened event log of concurrent activities in multi-threaded applications. To address that, OmegaLog intercepts all write syscalls on the host using a kernel module and identifies which write syscalls belong to application event logging using heuristics discussed in Section 5.4. After that it only appends the `PID/TID` of the process/thread that emitted the event and along with the timestamp of the event's occurrence to the identified log messages, generating enhanced event log messages.[1] Finally, OmegaLog uses Linux Audit API to add the enhanced event log message to the whole-system provenance log file, which provides an ordering for both application- and system-level events.

5.6 INVESTIGATION PHASE

Following an attack, an administrator can query OmegaLog's log parser and graph generator modules (⑤ in Fig. 5.5) to construct a UPG chronicling the system- and application-layer events related to the intrusion.

[1]Applications that make use of rsyslog facility [142] to write LMS is the one exception to the rule where LMS writing process's PID is not equal to the original application process that produced the LMS. However, in such case we can easily extract the `PID/TID` of original application process because rsyslog use well-defined message format [139] with PID added by default.

5.6.1 Universal Provenance

Given application binaries, whole-system provenance logs, and application event logs, during the investigation phase, we aim to generate a UPG while preserving the three properties of causality analysis. Algorithm 5.2 describes how to construct the backward-tracing UPG from the universal log file, specifically a backtrace query from an observable attack *symptom event*; the approach to building forward-trace graph follows naturally from this algorithm and is therefore omitted. When an application event log (an augmented LMS) is encountered while parsing the universal log (Function IsAppEntry in Algorithm 5.2), it is necessary to match the event to a known LMS for the application in our LMS paths. That matching is performed by the MatchLMS function as described below.

5.6.2 LMS State Matching

This procedure entails matching of a given runtime application log entry to its associated LMS in the LMS control flow paths DB. For each log entry in the universal log, the matcher identifies all LMS regexes that are candidate matches. For example, if the event message is

```
02/15/19 sshd [PID]: PAM: password from user root accepted
```

the matcher will look for substring matches, and this will solve the issue of identifying the actual application log entry from the preamble metadata, e.g., "02/15/19 sshd[PID]:".

Ranking LMS. An application log entry may match to multiple LMS regexes in the LMS path DB; this happens because of the prevalence of the %s format specifier in LMS, which can match anything. Therefore, OmegaLog performs a ranking of all the possible candidate matches. We use regex matching to identify the number of non-regex expressions (*i.e.*, constants) in each match. Going back to the example, "`PAM: password from user root accepted`" will match "`PAM: password from user .* accepted`" with a ranking of 5, which is equal to the number of non-regex word matches. Finally, the matcher will return the LMS that has the highest rank or the highest number of non-regex word matches that reflects the true state among the candidate LMSes.

State Machine Matching. Once the candidate LMS (LMS_{cand}) has been identified for an application log entry, OmegaLog attempts to match the LMS_{cand} to a valid LMS path in the database. If this is the first event message, we use a set of heuristics to figure out where we should start from. However, since the matching process can start anywhere in the applications lifetime, usually we have to resort to an exhaustive search over all nodes in the LMS control flow paths. Once we identified the starting node, we keep state in the parser

Algorithm 5.2: UPG Construction

Inputs : Universal log file L_{uni};
Symptom event e_s;
LMS control flow paths $Paths_{lms}$;
Output : Backward universal provenance graph G
Variables: $LMS_{state} \leftarrow$ Current state of LMS;
$eventUnit[Pid] \leftarrow$ events in current unit related to Pid;
$endUnit \leftarrow$ flag to partition execution into unit;

```
1  endUnit ← false
2  foreach event e ∈ L_uni happened before e_s do
3      if IsAppEntry(e) then
4          LMS_cand = GetLMSRegex(e)
5          endUnit = MatchLMS(LMS_cand, Paths_lms, LMS_state, eventUnit[Pid_e], L_uni)
6      end
7      if endUnit then
8          eventUnit[Pid_e].add(e)
9          Add all events from eventUnit[Pid_e] to G
10         endUnit ← false
11         eventUnit[Pid_e] ← null
12     end
13     else
14         eventUnit[Pid_e].add(e)
15     end
16 end
17 return G
```

that points to the possible transitions in the LMS paths graph. Upon the next log entry, we search the neighbors of the previous LMS for possible candidate matches. We rank those and return the one with the highest rank, and then advance the parser's state pointer. If OmegaLog cannot find a match in the neighboring LMS states, it advances to the lookahead and lookback matching steps.

Lookahead Matching. When the previous state in the LMS path is known, we may not find a match in a neighboring LMS state because for example (1) the application is running at a different log level, (2) OmegaLog missed the LMS corresponding to the log message in the static analysis phase (for example, the function might be inlined, or we could not concretize its values), or (3) the log message is coming from a third-party library. We therefore start looking deeper into the reachable states from the current parser state. If we find multiple candidates, we again rank them and return the one with the highest rank. If we do not find one, we then keep increasing the lookahead up until we hit a certain threshold that can be set at runtime. If we find a match, we move the parser to that state and repeat until we match a candidate LMS at the end of LMS control flow path. At that point, we set the *endUnit* flag to true.

As described in Section 5.4, in certain cases LMS may not be able to correctly partition the execution because there are syscalls after the loop-ending LMS or syscalls before loop-starting LMS. During offline analysis, OmegaLog marks such LMS and keep track of any syscalls that we should expect during runtime. If we observe such case during state matching process, we match those syscalls besides matching LMS and add those syscalls into the execution unit. Function MatchLMS in Algorithm 5.2 also handles such cases and appropriately sets the *endUnit* flag to true.

Lookback Matching. If the above lookahead step fails because we cannot find the end state in the LMS path, we try to search the heads of loops that are of the form (`while(1)`, `for(;;)`) in the LMS control flow path. The intuition behind this identification is that we might have hit the start of a new execution unit and thus we would need to restart from a new stage. If this fails, we perform an exhaustive search of LMS that can happen before the current state in the LMS paths using the same intuition mentioned before. If in either case, we get a match we set the *endUnit* flag to true. Note that fallback matching allows us to generate execution units even if we have only one log message at start or end of the loop, because we use the next execution unit's log message to partition the current execution unit.

5.7 EVALUATION

In this section, we evaluate OmegaLog to answer the following research questions (RQs):

RQ1 What is the cost of OmegaLog's static analysis routines when extracting logging information from binaries?

RQ2 How complete is our binary analysis in terms of finding all the LMSes in an application?

RQ3 What time and space overheads does OmegaLog impose at runtime, relative to a typical logging baseline?

RQ4 Is the universal provenance graph causally correct?

RQ5 How effective is OmegaLog at reconstructing attacks, relative to a typical causal analysis baseline?

Experimental Setup. We evaluated our approach against 18 real-world applications. We selected these applications from our pool of applications discussed in Section 5.2 based on popularity and category. Moreover, most of these applications were used in the evaluation of prior work on provenance tracking [12, 13, 16, 21]. For each program, we profile two

verbosity levels, `INFO` and `DEBUG`, when considering the above research questions. Workloads were generated for the applications in our dataset using the standard benchmarking tools such as Apache Benchmark `ab` [143] and FTPbench [144].

All tests were conducted on a server-class machine with an Intel Core(TM) i7-6700 CPU @ 3.40 GHz and 32 GB of memory, running Ubuntu 16.04. To collect whole-system provenance logs we used Linux Audit Module[2] with the following syscall ruleset: clone, close, creat, dup, dup2, dup3, execve, exit, exit_group, fork, open, openat, rename, renameat, unlink, unlinkat, vfork, connect, accept, accept4, bind. OmegaLog's offline algorithm accepts a single configuration parameter, `maxBackTrace`, that sets the maximum depth of symbolic execution operations. After experimenting with that parameter, we found that a value of 5 was enough to guarantee >95% coverage for 12 of the 18 applications we analyzed, as we discuss in the following section. In fact, our experiments have shown that we did not need to increase the symbolic execution depth beyond 3 basic blocks.

5.7.1 Static Analysis Performance

Table 5.3 shows how much time it takes to identify and concretize LMS from application binaries and subsequently generate LMS path models (Algorithm 5.1). We first note that the overhead of building the LMS paths (LMSPs) is reasonable for a one-time cost, taking 1–8 seconds for most applications, with a maximum of 3 minutes for `PostgreSQL`; the increase for `PostgreSQL` is due to the larger number of LMS paths captured by OmegaLog. On the other hand, average time to generate an LMS column shows the time to generate the `FastCFG` and concretize the LMS dominates OmegaLog's static analysis tasks, ranging from a minimum of a minute and a half (`Transmission`) to a maximum of 1.2 hours (`PostgreSQL`). Those two tasks are in fact highly dependent on Angr's raw performance. As acknowledged by the Angr tool developers [146], the static analyzer's performance is handicapped because it is written in the Python language with no official support for parallel execution.

Our results show no direct relationship between the size of the binary of the application being analyzed and the overall analysis time. By inspecting the applications' source code, we found that OmegaLog's performance is more informed by the structure of the code and the logging procedures. We can see intuitively that as the number of found callsites increases, the number of peephole symbolic execution steps needed also increases, thus increasing the total concretization time. However, that does not generalize to all the applications; for example, the analysis of `NGINX` (2044 KB binary) completed in 13 minutes concretizing 925 LMS while

[2]We make use of the Linux Audit framework in our implementation. However, our results are generalizable to other system logs, such as Windows ETW [55] and FreeBSD DTrace [145].

Table 5.3: Application logging behavior and performance results of OmegaLog's static analysis phase. EHL stands for event handling loop; IN+DE means that both INFO and DEBUG verbosity levels are present in the loop; LMSPs: Log message string paths; Callsites are identified log statements; and "Cov. %" denotes coverage percentage which is the percentage of concretized LMS to callsites.

Program	**Binary**	**Log Level**	**Avg. Time (sec)**		**Number of**		**Completeness**	
	Size (kB)	**inside EHL**	**LMS**	**LMSPs**	**LMS**	**LMSPs**	**Callsites**	**Cov. %**
Squid	64 250	IN+DE	831	46	64	157 829	70	91
PostgreSQL	22 299	IN+DE	3880	258	3530	4 713 072	5529	64
Redis	8296	INFO	495	7	375	34 690	394	95
HAProxy	4095	IN+DE	144	4	53	13 113	56	95
ntpd	3503	INFO	2602	4	490	10 314	518	95
OpenSSH	2959	IN+DE	734	4	845	11 422	869	97
NGINX	2044	IN+DE	775	11	923	8463	925	100
Httpd	1473	IN+DE	99	2	211	3910	211	100
Proftpd	1392	IN+DE	201	4	717	9899	718	100
Lighttpd	1212	INFO	1906	2	349	5304	358	97
CUPSD	1210	DEBUG	1426	3	531	4927	531	100
yafc	1007	IN+DE	88	2	57	3183	60	95
Transmission	930	IN+DE	102	2	178	5560	227	78
Postfix	900	INFO	97	3	96	2636	98	98
memcached	673	IN+DE	193	7	64	19 510	69	93
wget	559	INFO	200	3	84	3923	275	31
thttpd	105	N/A	157	8	4	14 847	5	80
skod	47	N/A	12	0	25	115	25	100

`Lighttpd` (1212 KB, almost half of `NGINX`'s binary size) required 32 minutes concretizing only 358 LMSes.

Upon closer investigation of `Lighttpd`'s source code, we found that format specifiers (and thus LMS) were often passed as structure members rather than as constant strings (which form the majority of LMS in the case of `NGINX`). That will trigger the backtracing behavior of the PeepholeConcretization algorithm in an attempt to concretize the values of the struct members, thus increasing the cost of the symbolic execution operations performed by Angr. Sample code snippet from `Lighttpd` that triggers such behavior is shown in Listing 5.1.

Listing 5.1: Sample code snippet from Lighttpd codebase

```
/* log function signature: /src/log.c */
int log_error_write (server *srv, const char *filename, unsigned int line,
        const char *fmt /* our tool looks for fmt */, ...)
/* format specifier passed as struct member: /src/config-glue.c */
if (con->conf.log_condition_handling) {
  log_error_write (srv, __FILE__, __LINE__, "dss",
      dc->context_ndx, /* the fmt argument */
      "(cached) result:",
       cond_result_to_string (caches[dc->context_ndx].result)); }
```

The cases of `Lighttpd` and `NGINX` highlight the unpredictability of runtime of OmegaLog's static analysis when only the binary size or the number of identified callsites is considered. Rather, the runtime depends on the structure of the code and the anatomy of the calls to the log functions.

5.7.2 Static Analysis Completeness

We report on OmegaLog's coverage ratio, which represents the percent of concretized LMS relative to the count of identified callsites to logging procedures. As shown in the last column of Table 5.3, OmegaLog's coverage is >95% for all the applications except `PostgreSQL`, `Transmission`, and `wget`. We disregard `thttpd` since it presents a small sample size in terms of LMS where OmegaLog only missed 1 LMS during concretization. That speaks to OmegaLog's ability to consistently obtain most of the required LMSes and build their corresponding LMS control flow paths. We show in our experiments, this coverage ratio is sufficient to enable OmegaLog to perform execution partitioning and aid the investigation process without loss of precision. In addition, when LMSes are missing, OmegaLog's runtime parser can handle missing log messages through lookahead and lookback techniques. If OmegaLog fails to concretize an LMS, it is a reflection of the symbolic execution task's ability to resolve a format specifier for a logging procedure.

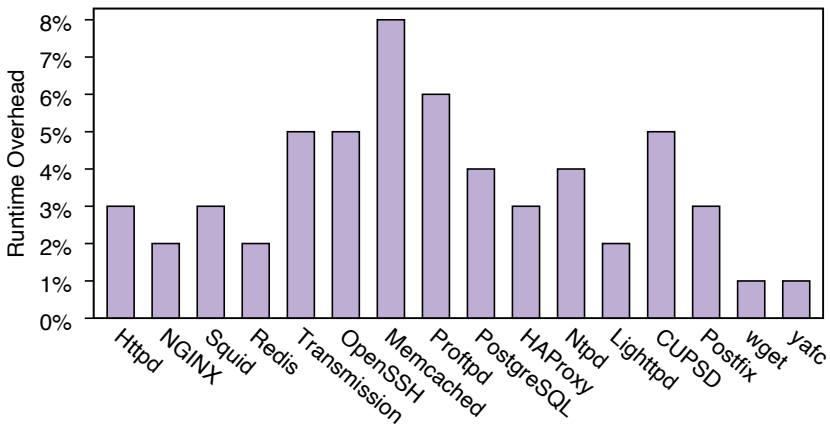


Figure 5.7: Runtime overhead for each applications in our dataset that has logging statement in the event-handling loop.

To better understand the conditions of OmegaLog's performance, we analyzed the source code of `PostgreSQL`, `Transmission`, and `wget` (64%, 78%, and 31% coverage, respectively). Our analysis revealed that in all three cases, symbolic execution was failing for logging procedures that use GNU's `gettext` for internalization (called using the "_" operator), as shown in Listing 5.2 below:

Listing 5.2: Part of logging procedure that uses GNU's `gettext` for internalization

```
/* Below code from Transmission: / libtransmission /rpc-server.c */
tr_logAddNamedError(MY_NAME, _("Couldn't find settings key \"%s\""), str);
/* Below code from wget: /src/convert.c */
 logprintf  (LOG_VERBOSE, _("Converting links in %s... "),  file );
/* Below code from PostGreSQL: /src/backend/commands/tablecmds.c */
default :
        /* shouldn't get here, add all  necessary  cases  above */
        msg = _("\"%s\" is of the wrong type");
        break; }
```

Since `gettext` is loaded dynamically as a shared library, Angr is not able to handle it appropriately during symbolic execution and cannot extract its return value, thus causing the failure of LMS extraction during the peephole concretization step. To confirm our findings, we reran the static analysis for `wget` and `Transmission` with the calls to `gettext` removed and were able to achieve coverage of 98.18% and 96.03%, respectively. One approach to addressing that issue using Angr would be to add hooks for all of `gettext`'s methods and return the arguments without changes. That would in turn provide Angr's symbolic execution engine with the arguments for concretization. We plan to address the issue in future work.

5.7.3 Runtime & Space Overhead

We measured the runtime overhead of OmegaLog compared to a baseline of application event log collection at the `INFO` and `DEBUG` verbosity with Linux Audit running. We turn on `INFO` and `DEBUG` level based on the application's logging behaviour required for execution partitioning. As shown in Figure 5.7, OmegaLog's average runtime overhead was 4% for all the applications that had logging inside the event-handling loop. Some applications, such as Memcached and Proftpd, exhibit high overhead because they are write-intensive applications; since OmegaLog intercepts every write syscall to disambiguate `PID/TID`, we expect to see higher runtime costs here. However, we argue that the benefits of OmegaLog for forensic analysis already justify the cost, and will consider alternative methods for process disambiguation in future work.

OmegaLog incurs space overhead because it records the `PID/TID` and timestamp for each application event message so that it can match the event to the appropriate system-layer task. At most, that addition requires 12 bytes per LMS entry. Our experiments confirm that the cost is negligible during typical use. For example, each unenhanced event message in NGINX is approximately 8.6 kB. If an NGINX server received 1 million requests per day and each request generated one event, the original event log would be 860 MB and OmegaLog would add just 12 MB to that total, around 1% space overhead.

5.7.4 Correctness of Universal Provenance Graph

OmegaLog modifies the whole-system provenance graph by adding *app log vertices* to generate semantic-aware and execution-partitioned universal provenance graphs. We describe three causal graph properties in Section 5.3 that the universal provenance graph needs to preserve for correct forensic analysis. To ensure the *Validity* property, we augment LMS with PID/TID information along with timestamps during the runtime phase so that we can causally associate application log vertices with process vertices in the whole-system provenance graph. To ensure the *Soundness* property, OmegaLog augments LMS with timestamps from the same system clock as the whole-system provenance graph and uses this timestamp as an annotation from process vertices to application log vertices. That edge annotation allows OmegaLog to respect the happens-before relationships while doing backward and forward tracing on the graph. Finally, since universal provenance graphs do not remove any causally connected vertices (besides false provenance introduced by dependency explosion in a manner consistent with previous work [13, 21]) we achieve the property of *Completeness*.

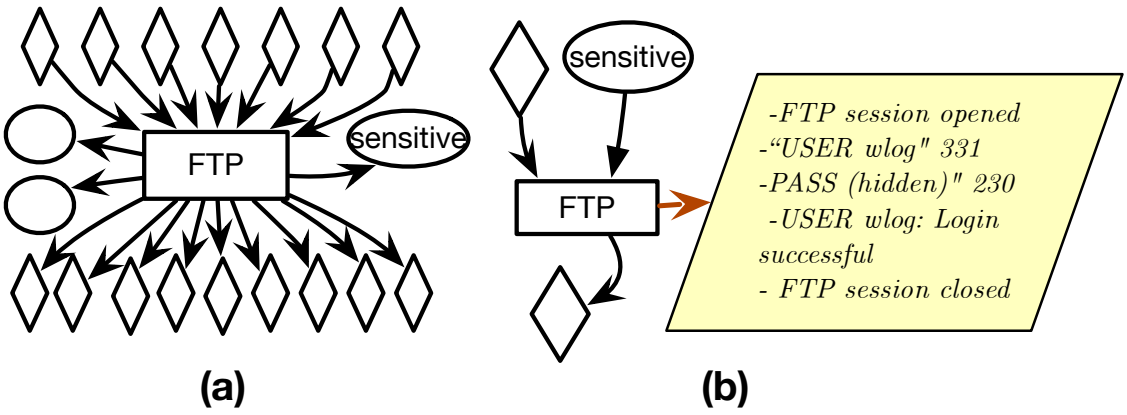


Figure 5.8: Information theft attack scenario. (a) Provenance graph generated using a traditional solution, which led to a dependency explosion problem with no semantic information. (b) Concise provenance graph generated using OmegaLog with semantic information.

5.7.5 Attack Investigation

We now evaluate OmegaLog's ability to aid in a typical attack investigation. To do so, we consider two additional scenarios as case studies. For each attack scenario, we manually verified its UPG to check that it preserved the three causality analysis properties that we discussed in Section 5.3. We note that the result that we presented in the motivating scenario (Section 5.1) was also procedurally generated using OmegaLog.

Information Theft Attack

An administrator made a mistake when configuring an FTP server, allowing users to read and transfer sensitive files from the server's directories. The issue was identified after several days, but the administrator now needs to identify which files were leaked, if any, to ensure that company secrets are safe. Using the sensitive files as a *symptom*, the administrator runs a backtrace query.

Figure 5.8(a) shows the attack investigation results using a traditional causal analysis solution, which confirms that the sensitive file was accessed. However, because of dependency explosion, it is impossible to determine who accessed the file and where it was transferred to. In contrast, Figure 5.8(b) shows the universal provenance graph produced by OmegaLog.

OmegaLog was able to partition the server into individual units of work based on event log analysis, removing the dependency explosion and identifying an IP address to which the sensitive file was downloaded. However, that information may not prove precise enough to attribute the attack to a particular employee or remote agent; fortunately, because Omega-

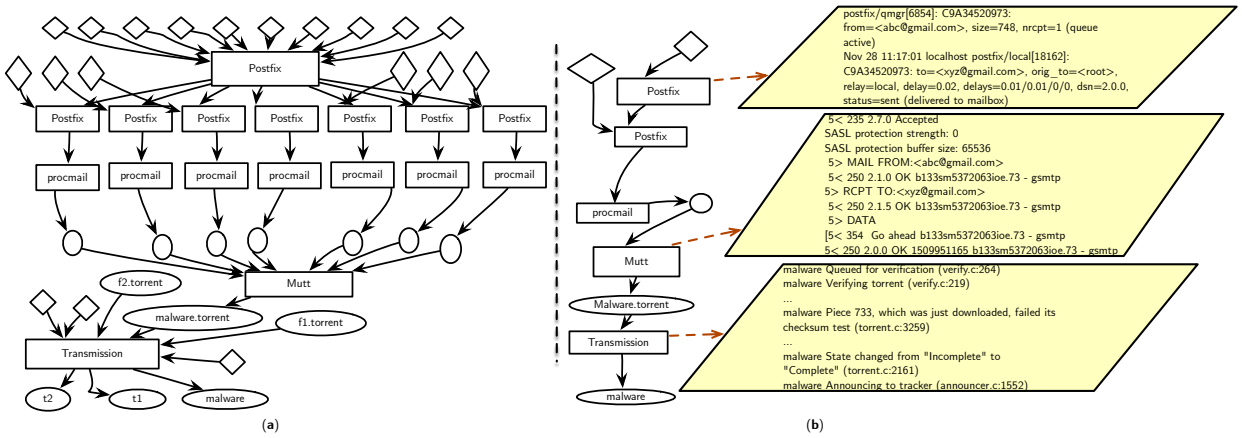


Figure 5.9: Phishing email attack scenario. **(a)** Attack provenance graph generated by traditional solutions. **(b)** Semantic-aware and execution-partitioned provenance graph generated by OmegaLog.

Log was able to associate the causal graph with event messages from the FTP server, the administrator is able to attribute the theft to a specific set of user credentials.

Note that while existing execution-partitioning systems such as ProTracer [21] and BEEP [13] could eliminate dependency explosion in this scenario, they would not enable user-level attribution of the attack.

Phishing Email

An employee uses the Mutt email client to send and receive personal emails on a BYOD workstation. One day, the employee receives a phishing email that offers a torrent for downloading a blockbuster movie. Employee opens the email, downloads the attached `.torrent` file. After that employee, used Transmission application to download the purported movie torrent file. Finally, employee opens the downloaded movie file but the file is actually malware that establishes a backdoor on the machine.

An administrator later notices that a suspicious program is running on the workstation and initiates forensic analysis to identify its origin. Figure 5.9(a) shows the causal graph that the investigation would yield based on simple `auditd`. As can be seen in the graph, the employee has actually opened three `.torrent` files with transmission-daemon. It is impossible to determine which `.torrent` input file led to the malware download. Even if out-of-band knowledge is used to identify the malicious torrent, the administrator will still be unable to trace back to the phishing email.

Figure 5.9(b) shows the UPG produced by OmegaLog. Because OmegaLog successfully

partitioned the Postfix and Transmission processes, the graph does not exhibit dependency explosion, making it easy to trace from the suspicious process back to the phishing email. Further, the OmegaLog graph provides additional documentation of application semantics, such as the email address of the sender, which may help the administrator correlate this attack with other intrusions. Such evidence cannot be provided by existing provenance trackers.

5.8 DISCUSSION

CFI Assumption. Control flow integrity (CFI) assumption is a limitation of OmegaLog; in fact, this is a big problem for almost the entirety of recent work in provenance-based forensic analysis space [10, 11, 12, 13, 14, 15, 16, 18, 21, 89, 106, 107, 108, 127]. OmegaLog assumes CFI of program execution because violation of CFI makes it impossible to give assertions about the trace logs of program execution. For example, execution units emitted from BEEP system [13] can not be trusted because an attacker can hijack control flow of the running application to emit misleading boundaries, confusing the investigator. Moreover, violations of CFI assumption enables post-mortem tampering of audit logs or even runtime control flow bending that causes misleading application event records to be emitted. Even though the main focus of our study is improving forensic analysis and solving CFI problem is ultimately an orthogonal problem to our study but we envision that future work on provenance will cater CFI violation problem for accurate forensic analysis.

Results Generalization. Provided that an underlying binary analysis tool has generated a reasonably accurate CFG, there are two considerations when one is evaluating the generality of OmegaLog. The first is whether or not the application being profiled includes logging events at key positions in the CFG such as the event handling loop. Our survey in Section 5.2 demonstrates that this is the case for mature open source client-server applications. The second consideration is whether the event logging statements are correctly identified and extracted by OmegaLog. Our evaluation (Section 5.7) demonstrated that we are able to identify log statements in all the profiled applications based on our heuristics for event-logging extraction.

OmegaLog assumes at least one log message printed in the event-handling loop to partition execution. OmegaLog uses ordered log messages in the universal provenance logs as a way to partition syscalls and make unit boundaries. Such an assumption only works for the applications that use synchronous I/O programming model. For instance, if an application is using asynchronous I/O and only prints one log message at the end of the event-handling

loop then concurrent requests will generate multiple syscalls without immediately printing log message at the end of each request. In such case, OmegaLog will not be able to correctly partition each request. One approach to solve this problem is to generate complete syscall mapping along with LMS paths model inside the event-handling loop during offline analysis and use this mapping to divide execution. We leave that as our future work.

Malware Binaries. Malware binaries may not produce any of the application logs that are required for execution partitioning. In that case, OmegaLog treats the whole malware execution as one unit and does not provide execution partitioning. That is acceptable since every output and input event from malware execution is important in forensic analysis.

CHAPTER 6: ZEEK AGENT: CORRELATING HOST AND NETWORK LOGS FOR BETTER FORENSICS

6.1 INTRODUCTION

In the previous chapters, we showed that host (system) logs are extremely useful during threat alert investigations. Based on kernel-layer information, host logs lack comprehensive network connection information, such as HTTP sessions with their requested URIs, key headers, and server responses; DNS requests; SSL certificates; SMTP sessions and SSL certificates. Such network-level information is often necessary to causally relate attacks that span across multiple hosts in the enterprise and generate complete contextual history for accurate investigations.

Even though correlating host and network logs can dramatically improve threat investigation capabilities, these two sources of logs are usually siloed in enterprises, making it difficult for security analysts to correlate them. Security analysts usually write long ad-hoc queries to correlate these logs, provided that they have the experience and expertise to do so. This correlation problem is further exacerbated by the sheer volume of host and network logs as analysts spend hundreds to thousands of employee hours manually stitching together these enormous logs.

To address this challenge and enable cross-log causal analysis, we designed an open-source endpoint monitoring tool called Zeek Agent [34]. Zeek Agent provides deeper visibility into enterprise-wide activities through transparently observing endpoint activities and correlating them with network logs in real-time. Zeek Agent consists of three components: 1) Zeek Agent Daemon – a high-performance and low-footprint host monitoring daemon that captures host logs using default auditing frameworks that come with the operating systems, 2) Zeek Agent Manager – a collection of Zeek scripts that allows security analysts periodically fetch host events across the entire enterprise and correlate those host events with the network flows present in Zeek logs, and 3) Zeek Agent Visualizer – a web-based graph visualization tool that allows analysts to construct, explore, and manipulate provenance graphs interactively.

Zeek Agent is inspired by the `zeek-osquery` system, which was proposed by Haas et al. [147] to correlate host logs and Zeek network logs. However, `zeek-osquery` was difficult to configure, manage, and extend [148] due to its dependence on OSQuery[149] to collect host logs. OSQuery is a large project that provides many more functionalities, such as profiling and API monitoring, than needed for endpoint security monitoring. To support so many monitoring features, OSQuery depends on a lot of packages (e.g., RocksDB) and requires setting up many configuration values. This made `zeek-osquery` project difficult to

maintain for long-term development. To solve this issue and provide a standalone endpoint security monitoring framework that is modular, extensible, and seamlessly integrates with the Zeek ecosystem, we implemented Zeek Agent.

Zeek Agent Daemon captures host process, file, and socket events and stores them in tables as a relational database similar to OSQuery [149], enabling security analysts to retrieve host events from the enterprise using SQL queries. On Linux, host monitoring data is captured using the Linux Audit framework [56]. On macOS, Zeek Agent leverages the Endpoint Security Framework [150] to capture file and process events. To collect socket events on macOS, Zeek Agent uses OpenBSM [151].

Security analysts can schedule SQL queries using Zeek Agent Manager to periodically fetch host events and store them along with Zeek network logs. We extended Zeek's `conn.log` file that enables security analysts to attribute network flows present in the `conn.log` file to the originating host processes.

Finally, we designed Zeek Agent Visualizer – a graphical user interface to aid the visualization and exploration of a unified provenance graph generated from the correlated host and network logs. This graph combines the causal reasoning strengths of the host log with the rich network information from the network logs, fundamentally increasing the power and applicability of either log types. Our tool features several new graph manipulation functionalities, such as real-time filtering, collapsing, and merging of facts, which allow security analysts to quickly drill down to relevant details and provide a better visualization experience. Note that our tool does not lose information so that the security analysts can revert any changes made during the graph manipulation. Most notably, our tool does not require security analysts to have any external skills, such as SQL and Cypher query language proficiencies, to generate a provenance graph.

We evaluated Zeek Agent for performance, scalability, and efficacy. Our evaluation results show that Zeek Agent exhibits a low runtime overhead on end hosts. To perform scalability analysis, we deployed in Zeek Agent in a cluster setting with a varying number of hosts. Using DARPA OpTC [152] attack dataset, we show that our graph visualization tool dramatically improves the threat alert investigation capabilities.

6.2 DESIGN

6.2.1 Overview

zeekagent is a modular and dynamic endpoint monitoring framework that seamlessly integrates with Zeek. At a high level, it aims to correlate host and network logs in a large

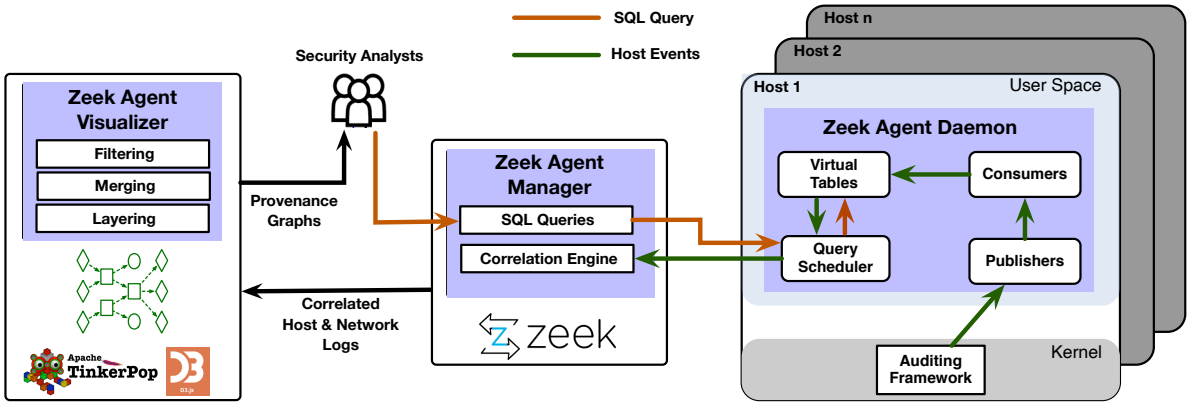


Figure 6.1: Zeek Agent workflow to collect hosts event logs and then correlate those logs with the network flows present in the Zeek logs. Our provenance graph visualization tool allows analysts to interactively manipulate and explore provenance graphs generated from the correlated host and network log.

enterprise and generate a unified provenance graph to accelerate the threat investigation process. The challenges mentioned in Section 6.1 motivate the following design goals of Zeek Agent:

G1 *Low Overhead.* Zeek Agent should have low runtime overhead to provide continuous (always on) security monitoring.

G2 *Non-intrusive.* Our system should not require any end system change or any kernel instrumentation.

G3 *Integrated.* Our system must correlate host and network logs in a manner that enables security analysts to perform cross-log causal analysis and generate a complete explanation of enterprise-wide activities.

G4 *Support Dynamic Monitoring.* Security analysts should be able to dynamically collect and filter host events without any downtime or reboots using Zeek Agent.

G5 *Interactive Graph Visualization.* Zeek Agent should provide an interface to interactively explore large provenance graphs generated from the correlated host and network logs, which will allow analysts to quickly drill down to relevant details during investigations.

The general workflow of Zeek Agent is shown in Figure 6.1. First, Zeek Agent Daemon collects syscall events on the end host and stores them into SQL virtual tables. After that, Zeek Agent Manager connects with Zeek Agent Daemon to schedule SQL queries written by

the analysts. Zeek Agent Manager retrieves host events, which are the results of SQL queries, and correlates those host events with the network flows present in the Zeek logs. Finally, Zeek Agent Visualizer displays a unified provenance graph generated from the correlated host and network logs for threat alert investigation.

Why SQL. The reason behind using SQL to introspect the operating system (OS) state of the hosts is as follows: SQL allows Zeek Agent to use a single query to collect host events from the entire enterprise. It does not matter if different hosts are running different OSes because the core OS concepts are common among different operating systems. For example, processes, files, and sockets abstraction are similar among Linux, macOS, and Windows. Not only that, but the attributes of those core concepts are also similar across operating systems, such as `pid` and `tid` from process events. For instance, consider the following SQL query:

```
select pid, exe_name, syscall from process_events
where  syscall != fork;
```

In this example, Zeek Agent will be able to fetch `pid`, `exe_path`, and `syscall` name from `process_event` tables from different hosts. In this example, we also have a constraint that excludes all the `fork` syscall events. As we can see from this example, we do not have to specify the underlying operating system type to fetch these events.

6.2.2 Zeek Agent Daemon

The main job of Zeek Agent Daemon is to efficiently collect and store host events for threat investigation. It consists of four sub-components, which are described below:

Publishers. The role of the publisher is to leverage the syscall monitor that comes default with the underlying operating system to collect events related to processes, files, and sockets. This allows Zeek Agent to fetch OS events in a non-intrusive manner, avoiding instrumentation or kernel modification on the end host. For the Linux platform, Zeek Agent Daemon uses Linux Auditd [56], while for macOS, it uses Endpoint Security Framework [150] and OpenBSM [151].

Consumers. The consumers' job is to retrieve events from publishers' queues. These events are then stored in different in-memory virtual SQLite tables based on the event type (e.g., file events). These virtual tables will allow Zeek Agent users to easily extend Zeek Agent Daemon to collect new event types on the host. Events stored in those virtual tables will be continuously pulled by Zeek using scheduled queries.

The publisher-consumer model has two advantages. First, this model enables better scalability by running multiple publishers and consumers in parallel and by caching messages. For example, currently, Zeek Agent Daemon supports the Linux Auditd, Endpoint Security and OpenBSM publishers and consumers. Second, this model allows users to extend Zeek Agent with new publishers and consumers without changing existing publishers and consumers. For example, new publishers and consumers can be added without making changes to the existing ones.

Virtual Tables. As mentioned above, consumers' job is to store host events in SQL virtual tables. A virtual table is a logical database that is not physically stored on a disk but allows users to run SQL queries without dealing with the complexities of tunning query performance. Users of Zeek Agent can set the size of the virtual table in the configuration file. During runtime, if the host events fill the whole virtual table before being pulled by Zeek, then Zeek Agent Daemon will drop those events in a FIFO fashion.

Currently, Zeek Agent supports three kinds of host event tables, which are described below:

- Host Process Event Table. To build process events table, Zeek Agent monitors `clone`, `fork`, `exec`, `exit`, and `exit group` syscalls. From these syscalls, we extract pid, parent pid, uid, image path of process executing, command-line arguments passed, and return value of the syscall.

- Host File Event Table. To populate the file events table, Zeek Agent subscribes to `open`, `open at`, `create`, `write`, and `close` syscalls. We extract pid, parent pid, uid, image path, file path, and inode number from these file-related syscalls.

- Host Socket Event Table. Finally, to build the socket events table, Zeek Agent uses `connect` and `bind` syscalls. From these socket-related syscalls, we extract pid, uid, image path, file descriptor, protocol family, local IP address and port and remote IP address and port.

Zeek Agent Daemon can be configured to exclude certain syscalls from being captured, at will, by specifying those syscalls within the Zeek Agent Daemon configuration file.

Query Scheduler. The query scheduler receives SQL queries from Zeek Agent Manager and executes them to extract the required fields from the required tables. An example SQL query is shown in section 6.2.1. Once this query is received, the query scheduler decides which table to extract records from, the fields of that table to include in all the records and the particular records to extract from that table based upon the conditions specified in the

SQL query. For example, in the example query shown, `pid`, `exe_path`, and `syscall` name from `process_event` tables are being extracted, excluding all fork syscall records. These records are then sent back to the Zeek Agent Manager.

There are two types of queries that the query scheduler deals with: one-shot queries and scheduled queries. One-shot queries, as the name implies, are queries that are executed by the scheduler only once, where as scheduled queries are executed repeatedly after the time interval sepcified. For example, if the specified time interval for a scheduled query is two seconds, that query will be executed repeatedly every two seconds, and the results of each individual execution will be sent back to the Zeek Agent Manager.

6.2.3 Zeek Agent Manager

Zeek Agent Manager is a collection of Zeek scripts to expose API for security analysts to write SQL queries, periodically fetch host events by running SQL queries, and correlate host socket events with the network flows present in Zeek logs. In this paper, we use the term *network flow* to describe an entry in Zeek's `conn.log`, which is a 5-tuple of local IP address, local port, remote IP address, remote port, and the protocol. We also denote the network flow initiator host as *originator* and the other receiving host as *responder*.

In order to communicate with the hosts and retrieve events, we leverage Zeek publish-subscribe library called Broker[1]. Similar to the `zeek-osquery` [147] system, Zeek Agent also allows custom groups of hosts so that security analysts can run certain SQL queries for a certain group of hosts. For example, suppose a security analyst wants to periodically fetch file events from all the research servers. In that case, security analysts can create a group of those research servers and then run SQL queries only on that group. Moreover, users of Zeek Agent can set the interval between scheduled queries on the hosts by setting the `default_query_interval` parameter in the configuration settings.

Attributing Network Flows to Host Socket Events. Once host events arrive at the node running Zeek Agent Manager, we correlate host socket events with the network flows present in Zeek's `conn.log`. We try to attribute both the originator and the responder of a network flow to their host socket events if both hosts are in our enterprise network.

In order to this attribution, we match 5-tuple from network flow with the 5-tuple present in the host socket event. However, the event data obtained from auditing frameworks, such as Linux Audit and OpenBSM, contain partial information, i.e., some elements are missing from host socket event 5-tuple. For example, the host socket event generated from `connect`

[1] `https://docs.zeek.org/projects/broker/en/current/`

syscall does not contain the local address and local port in the 5-tuple. Due to this partial information, we will have ambiguity when we correlate host socket events with network flows inside Zeek, i.e., one single network flow attributed to multiple host socket events.

One possible solution to this issue will be to use `procfs` and fetch that missing information. However, such a solution can introduce race conditions while probing, leading to incorrect results. Therefore, we currently do not use this solution.

To do this correlation or attribution inside Zeek script land, we use the methodology described in `zeek-osquery` paper. We add a handler for `connection_state_remove()` event in Zeek script land. Inside that handler, we perform the following three steps to attribute network flows to host socket events.

- Given a network flow, we extract the local IP address and the remote IP address from that flow. Using the local IP address and remote IP address, we try to figure out if these IP addresses belong to hosts in our enterprise network where Zeek Agent is deployed. This requires us to maintain a state of IP addresses of all hosts in our enterprise. In Zeek script land, we use a `table` to store this state.

- Then, on the originator side, we match the remote IP addresses of network flow and host socket event, while on the responding side, we match the local IP addresses from network flow and host socket event.

- Finally, on the originator side, we match the local IP address of network flow with the local IP address of the host socket event, while on the responding side, we match the remote IP address of network flow with the remote IP address present in the host socket event.

By performing all three steps in the above list, our correlation or attribution results will be unambiguous, i.e., one network flow attributed to exactly one originator socket and one responder socket event. However, the third step is only possible if we have all the information in the host socket event 5-tuple. Since we currently do not have all the information, our correlation results are vague. In the case of vague correlation, we list all possible host socket events with the network flow under analysis. Note that if two hosts with a `connect` syscall to the same remote IP address and port, our correlation will be unambiguous because of Step one. But, our correlation will be ambiguous for the same host with multiple network flows to the same remote IP and port (from different source ports).

Once we have correlated network flows with host socket events, we add `orig_seuid` (a socket identifier for originator) and `resp_seuid` (a socket identifier for responder) columns

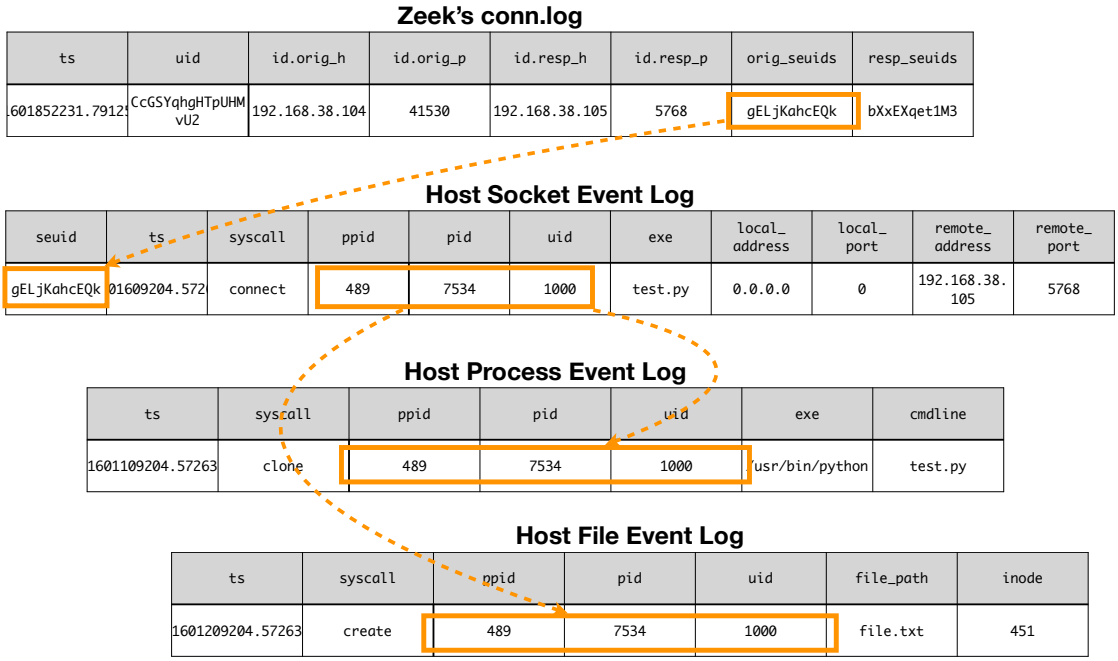


Figure 6.2: Example of correlating host and network log event table entries.

in Zeek's conn.log. These uids link network flows with the host socket event log as shown in Figure 6.2. From the host socket event log file we can find out the process responsible for this network flow. Using this process pid, parent pid, and uid, we can link it to the host process events log to find out the command line arguments and other attributes of this process. Finally, we can also look into the host file event log file to see if that process created or opened any files.

Grouping functionality. In addition to scheduling queries for individual hosts, the Zeek Agent Manager also allows us to assign a query to a group of Zeek Agent hosts. The groups are made using a subnet mask. The user provides a subnet along with the query to be scheduled. The Zeek Agent Manager then matches the subnet against the ip address of each of the host and the ip addresses which falls under the subnet are added to the group. A host can be a part of many different groups at the same time. All the hosts within a group executes the same query periodically and send the results back to Zeek Agent Manager.

Scalability enhancements. Inorder to ensure that Zeek Agent Manager is scalable, a number of functionalities were implemented. Firstly, all the socket events which have been successfully attributed are removed from socket events table. This ensures that the table size does not become too large as socket events come in. Secondly, the framework makes sure that whenever a Zeek Agent sends an exit process syscall then the state for that process is removed from the memory. This helps to ensure that Zeek Agent Manager uses memory

efficently and its memory footprint scales linearly with the number of Zeek Agent hosts.

6.2.4 Zeek Agent Visualizer

Provenance graphs generated from host and network logs are usually enormous in size, with thousands or even millions of vertices. Such large provenance graphs can overwhelm the security analysis and undermine the whole threat investigation process. Although there are several provenance graph visualization tools [10, 14, 20, 106] that exist in the literature, they statically generate a provenance graph and lack graph manipulation features. Moreover, these systems do not scale beyond several hundred vertices. Provenance Explorer [153] and Prov Viewer [154] provide interactive provenance graph visualization capabilities; however, these systems are not designed to construct provenance graphs from kernel-level audit logs.

To overcome this challenge, we designed and implemented a web-based graph visualization tool that allows security analysts to construct, explore, and manipulate provenance graphs interactively for better threat alert investigation. We implemented a web-based graphical user interface using D3.js[2] for our tool. By using a web-based interface, we removed any dependence on external packages and avoided software update overheads. We have used elasticsearch as our database. The reason why we chose elasticsearch as our database is because it can execute complex queries really fast. Moreover, it can scale horizontally, which is very important when dealing with a large dataset that can further increase in size. Furthermore, it stores data as structured JSON documents, and since the log files generated by Zeek Agent are already in JSON format, almost no preprocessing is required on data from the logs. Our novel provenance graph visualization tool features several new functionalities that not only accelerates the threat investigation process but also discovers other compromises that analysts might not be aware of. Following are the main features of our Visualizer (not in the order of importance).

Vertex, edge information, and exploration. Clicking on a vertex highlights it as an active vertex and displays its children and parent vertices (if any), as well as display its information. The edge connecting two vertices also has an edge label showing the relation between the vertices. The graph is mostly explored in such a way that the parent vertex is displayed on the left side of the clicked vertex, and the child vertex is displayed on the right side of the clicked vertex. This helps to see the chain of events in time from left to right.

Filtering. Our visualization tool allows a security analyst to filter vertices from the generated provenance graph based on labels (process, file, socket, zeek, network). This ability

[2]https://d3js.org/

enables security analysts to remove irrelevant information for a given threat investigation and focus on the most important parts of the provenance graph. Filtering hides the filtered vertices which are still explored in the background in their "hidden state" as the graph exploration progresses. This helps the analysts to view all the hidden vertices in their new state at any point in time if they choose to unhide them.

Merging similar file vertices. We merge similar file vertices into a single vertex. The criteria for merging similar file vertices is that if all the properties of file vertices are the same except the timestamp (the time at which a file operation was done on the file) we consider the vertices to be similar and therefore merge them into one vertex. This significantly reduces the size of the graph without affecting the correctness of the causal analysis. For example, an attacker may write to an existing file several times, for an analyst even if the file has been written once, it is flagged as a compromised file and therefore by merging such vertices into one, we have significantly reduced the size of the graph.

Merging similar socket vertices. We provide a checkbox to interactively merge similar socket vertices into a single vertex. We consider socket vertices similar if their remote IP addresses are the same and they are connected to the same process vertex. This significantly reduces the size of the graph without affecting the correctness of causal analysis, as shown by previous studies [30, 31].

Layers of visualization. Our visualization tool uses the concept of layers of visualization, which aims to put a spotlight on certain subgraphs rather than displaying the whole graph during exploration. In this idea, if a user clicks a vertex, our tool will show its neighbors, expanding new paths for further exploration. While if a vertex is not clicked, then that vertex will slowly vanish as we progress in the exploration. The way this works is that every time a vertex is clicked, it and its neighbouring vertices are marked as active vertices, and the edges connecting them are marked as active edges. These active vertices and edges will form layer 0 (the layer with the highest opacity). If a user then clicks on another vertex, the vertices and edges in layer 0 will be shifted to the next layer i.e layer 1 and will become old vertices and edges (if they are not the clicked vertex or the neighbours of the clicked vertex). The clicked vertex and its neighbouring vertices will be marked as active vertices, and the connecting edges as active edges and will become the new layer 0. The vertex will not be shifted to the next layer if it is still a neighbouring vertex of the clicked vertex, or the clicked vertex itself. When the user again clicks on some other vertex, the active vertices and edges will be marked as old vertices and old edges, and will be shifted from layer 0 to layer 1 (if not an active vertex) and the vertices and edges that were in layer 1 will be shifted from layer 1 to layer 2 and hence become older than before. The clicked vertex and its neighbouring

vertices and connecting edges will become active vertices and active edges of layer 0. The vertices and edges in a deeper layer will have a lower opacity than the layer preceding it. For example, vertices and edges in layer 3 will be more transparent than vertices and edges in layer 2.

Once the nth layer is reached, where n is the number of layers set by the user, the opacity of vertices and edges will be decreased to a level where they vanish. If the vertices completely vanish in the process, they can no longer be clicked, however, upon clicking a vertex that is vanishing, it and its neighbours become active and therefore bright again. For example, if a vertex that is in layer 2 is clicked, it and its neighbouring vertices and the connecting edges will become active and therefore will form the layer 0, while the previous layer 0 now becomes layer 1 with increased transparency. A vanished vertex can come back to life again in the same position it was vanished at, if any of its neighbour is clicked. Layering can therefore help provide a precise, local view rather than a global one. Our tool also allows the user to control how many new layers can be explored before vanishing old layers. Figure 6.3 illustrates this concept with seven layers.

Limit the number of vertices. Since provenance graphs of system logs have the potential to generate millions of new vertices, it can overwhelm an analyst if displayed on an interface. Therefore, an analyst can limit the number of entries that are displayed for each label as well as limit the number of entries that are fetched from the database where the logs are stored. Since, during merging, the size of the graph can be significantly reduced, therefore an analyst will find it more convenient to have separate limits on fetched vertices as well as displayed vertices.

Forward and backward tracking. The security analyst has the option to either only forward track, backward track or both when exploring. In forward tracking, only the children vertices will be explored and the graph will only move forward in time. In backward tracking only the parent vertices will be explored and the graph will only move backward in time, this is particularly useful when the user does not want exploration of children vertices and is only interested to go back in time, for example, to view the initial infection point (IIP). Since there may be many children vertices as compared to probably a single parent vertex, this will help prevent the visualizer to display the many children vertices when clicking on a vertex. At any point in time an analyst can enable forward tracking and click on the vertex to observe its children vertices and vice versa.

Advanced search. An analyst can search on unique id of each vertex, file name and PID. This will help provide the analyst a good reference point, for example a known compromised file, from which the graph can be explored in either direction (forward or backward tracing).

A search can be made on a specific label as well. Searching on files for example will display up to n number of files, where n is the display limit.

Pinning vertices. Security analysts can also set checkpoints by pinning a certain vertex, which disables a vertex from vanishing. This feature is extremely useful for investigating multistage APT attacks as security analysts can pin each attack stage while exploring.

Freeze exploration. Our Visualizer allows users to "freeze" the graph at any time. This allows the analyst to click on the vertices and see its description without making it explore and display its neighbouring vertices. This can help analysts to select relevant vertices for exploration, without exploring every vertex to view its description, thereby reducing the size of the graph significantly.

Dynamic positioning of vertices using d3.js. For the visualization, we took inspiration from physical forces that can help visualize the interactions between bodies (vertices and edges in our case), and therefore adopted d3's force layout to visualize the graphs. d3s force layout uses a physics-based simulator for positioning the visual elements of the graph. It simulates forces that allows a security analyst to control the position of vertices in relation to each other and the simulation. d3 forces allows vertices to attract or repel one another, and vertices can be configured to attract to the center of gravity. d3 force can calculate the velocity and position of incoming elements that are governed by rules mimicking the laws of physics. We have carefully chosen the ideal parameters of physical forces and rules and used a collision detection mechanism for the generated graph that can not only help find a stable equilibrium for the vertices during the simulation, but also prevent the new vertices from overlapping thereby visualizing stunning graphs. We also change the center of gravity at each iteration in such a way that the children of a vertex can fan in and out according to the parent vertex's position to prevent crowding of nodes in a certain area. Since the simulation is aimed at visualizing the new vertices (active vertices) that were not previously part of the graph. In order to prevent overlapping of these new vertices with the old ones, Zeek Agent Visualizer finds the closest positions for new vertices that have not been taken by any older vertex along the x-axis.

6.3 SECURITY ANALYSIS

Our arguments for satisfying the design goals mentioned in Section 6.2 are as follows:

Low Overhead (G1). Zeek Agent Daemonis implemented using C++ language and follows publisher-consumer model, which keeps the runtime overhead minimal on the end host. Moreover, we leverage SQLite virtual tables that are designed to provide faster execution of

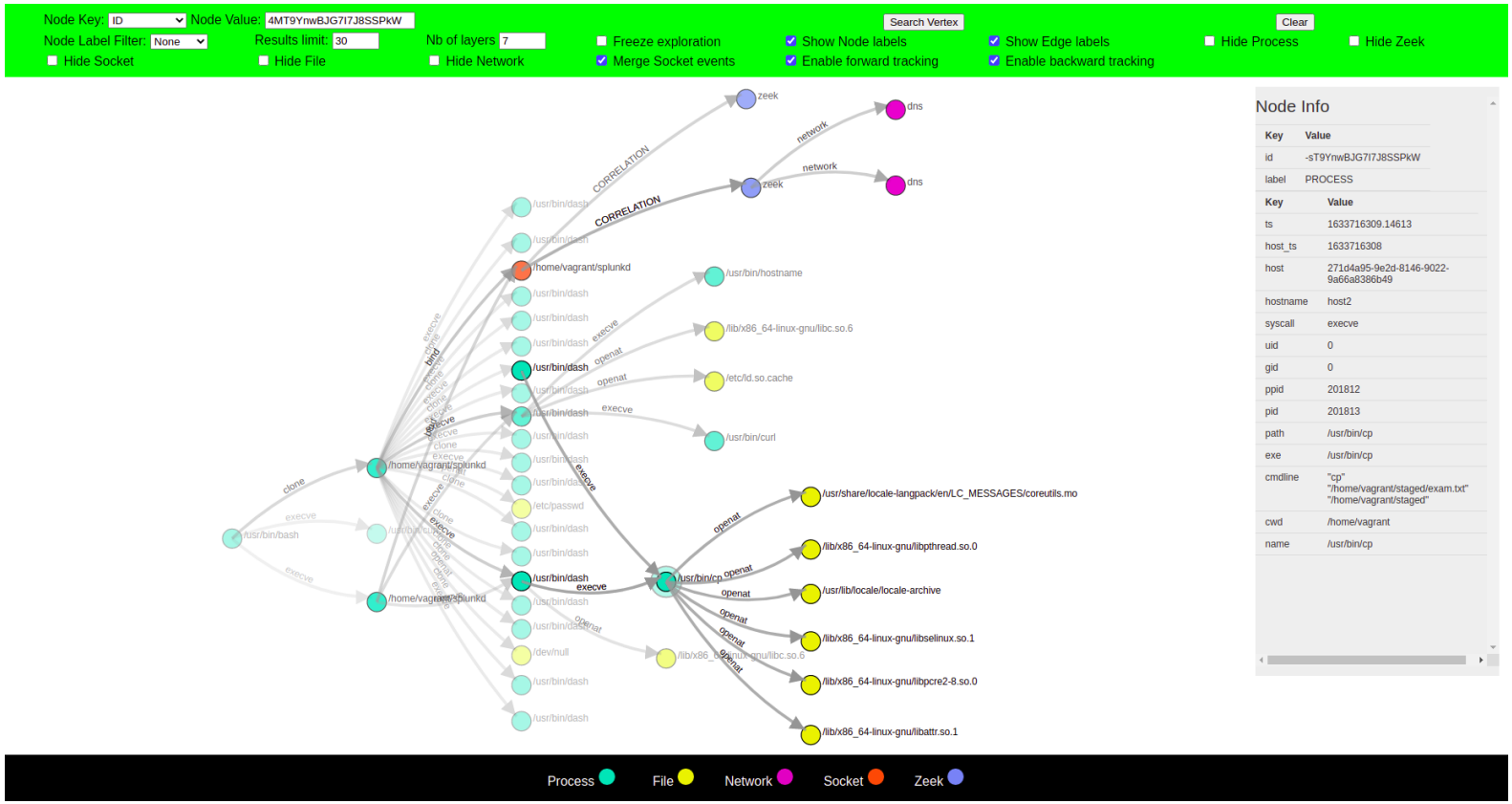


Figure 6.3: Provenance graph with seven visualization layers. The edges and vertices with higher transparency were clicked before the vertices and edges with lower transparency.

queries without excessive memory footprint.

Non-intrusive (G2). Zeek Agent Daemonleverages commodity auditing frameworks (e.g., Linux Audit) that come default with the operating systems to collect monitoring data. Moreover, our correlation methodology is oblivious to the underlying operating system.

Integrated (G3). In Zeek Agent we retrieve host events using Zeek's Broker Library, allowing us to process host events analogous to network events in Zeek. After that, we correlate host and network layers using the 3-step procedure described in Section 6.2.3 and extend Zeek's conn.log with the identifiers that link network flows to the host socket events. Thus, all layers share a common language to describe an activity.

Support Dynamic Monitoring (G4). Zeek Agent Manager enables security analysts to retrieve host events using SQL queries. Using this feature, the security analysts can *dynamically* select what to collect from the host based on their investigation needs.

Interactive Graph Visualization (G5). Given a threat alert event, Zeek Agent Visualizer seamlessly fetches the correlated host and network logs and constructs a provenance graph. Our web-based graphical interface enables security analysts to interactively explore different parts of the graph without overwhelming them with irrelevant information.

6.4 EVALUATION

Our evaluation seeks to answer the following questions: (1) what is the resourse usage of Zeek Agent? (2) How scalable is Zeek Agent Manager in large scale enterprises with increasing number of Zeek Agent Daemon hosts? (3) How effective and accurate is Zeek Agent as an investigation tool?

We conducted our evaluation on an Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz server with 16 cores and 64 GB RAM. VMs were provisioned with VirtualBox using a common deployment model of 2GB RAM and 2 vCPUs and ran Ubuntu 20.04.

In the first experiment the aim was to study how Zeek Agent Manager scales with increasing number of Zeek Agent Daemon processes. For this a central logger VM was made which ran Zeek Agent Manager and many host VMs were constructed running Zeek Agent Daemon process. The host VMs were turned on in multiples of two and the resource usage on logger machine was noted. The resource usage data included CPU and RAM utilization. During the experimentation, Zeek Agent Manager was fetching logs from all the Zeek Agent Daemon hosts in the cluster and performing correlation between host and network logs.

Furthermore, to generate activity on host VMs we simulated several real-world workloads

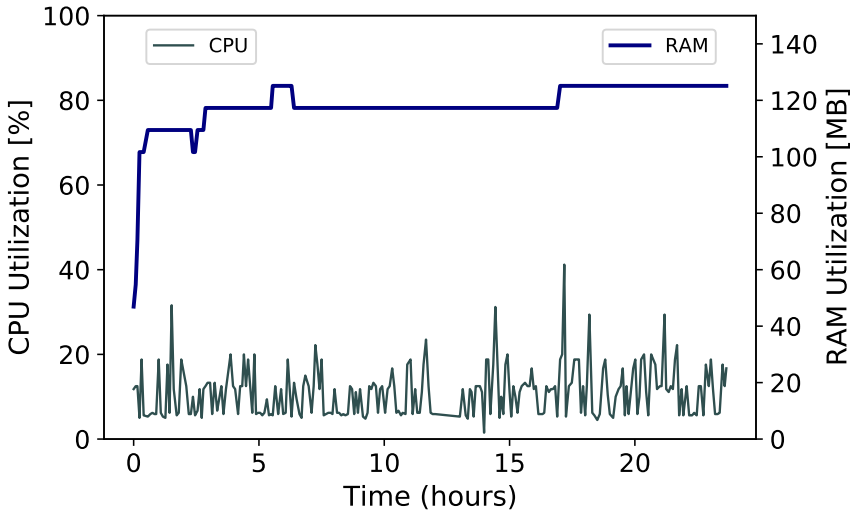


Figure 6.4: CPU and memory utilization of Zeek Agent Daemon process over the period of 24 hours while running several benchmarks.

using standard benchmarks such as LMBench[3] and Apache benchmark[4]. The benchmarks generated activity to profile the CPU, RAM and file system of the host VMs. The Zeek Agent Daemon was constantly collecting logs about these activities and was sending it to the Zeek Agent Manager node.

The second experiment was conducted to check the resource utilization on a single node running Zeek Agent Daemon process. For this CPU and RAM usage of Zeek Agent Daemon process was collected over a period of 24 hours and the results were plotted. In all this time, the benchmarks were also executing on the machine being tested.

6.4.1 Zeek Agent Daemon Resource Usage

After collecting the data for 24 hours, the results were plotted. Figure 6.4 shows our resource usage experiment results. We can see that resource usage of Zeek Agent Daemon was quite low. The max peak RAM utilization was just 120MB, and the average CPU usage was below 20%. These results show that even with resource intensive benhcmarks, the runtime overhead of Zeek Agent Daemon is low.

[3]Available at `http://lmbench.sourceforge.net/`
[4]Available at `https://httpd.apache.org/docs/2.4/programs/ab.html`

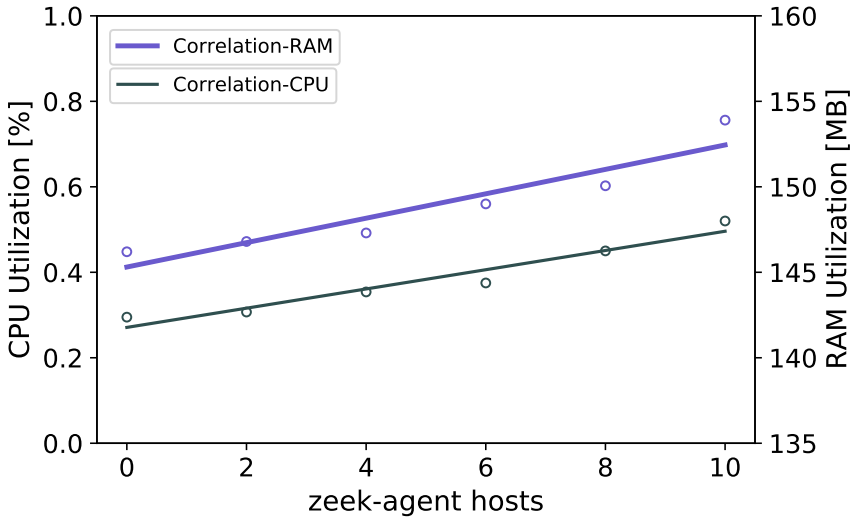


Figure 6.5: CPU and memory utilization of Zeek Agent Managerwith varying number of Zeek Agent Daemon hosts.

6.4.2 Zeek Agent Manager Scalability Analysis

The results for our scalability analysis are shown in Figure 6.5. We can see that the Zeek Agent Manager process scales linearly with the increasing number of Zeek Agent Daemon hosts. Both the RAM and CPU usage follows a linear trend with a gentle slope. This means that the resource usage of Zeek Agent Daemon process increases very reasonably when the workload from Zeek Agent Manager hosts increases.

6.4.3 Case Study: Malicious Upgrade

To show the effectiveness of correlating host and network logs and building a unified provenance graph, we leveraged DARPA Operationally Transparent Cyber (OpTC) Datasets [152] that includes both host events and Zeek network logs. This dataset contains more than 17 billion events collected from 500 host, running Windows 10 operating system. Beside benign activities, this dataset also includes malicious events from three APT attacks that were injected by red team over three day period in their 500 host network. We used day three APT, also known as Malicious Upagrade in the dataset documentation, for evaluating Zeek Agent correlation and graph visualization capabilities.

Attack Scenario. The day three APT attack begins by downloading Notepad plus software on one of the hosts called Sysclient0051. This software was susceptible to a malicious upgrade. When updated, the software contact a malicious server and downloaded a binary called `gup.exe`, which was essentially a Meterpreter payload. Meterpreter is a Metasploit

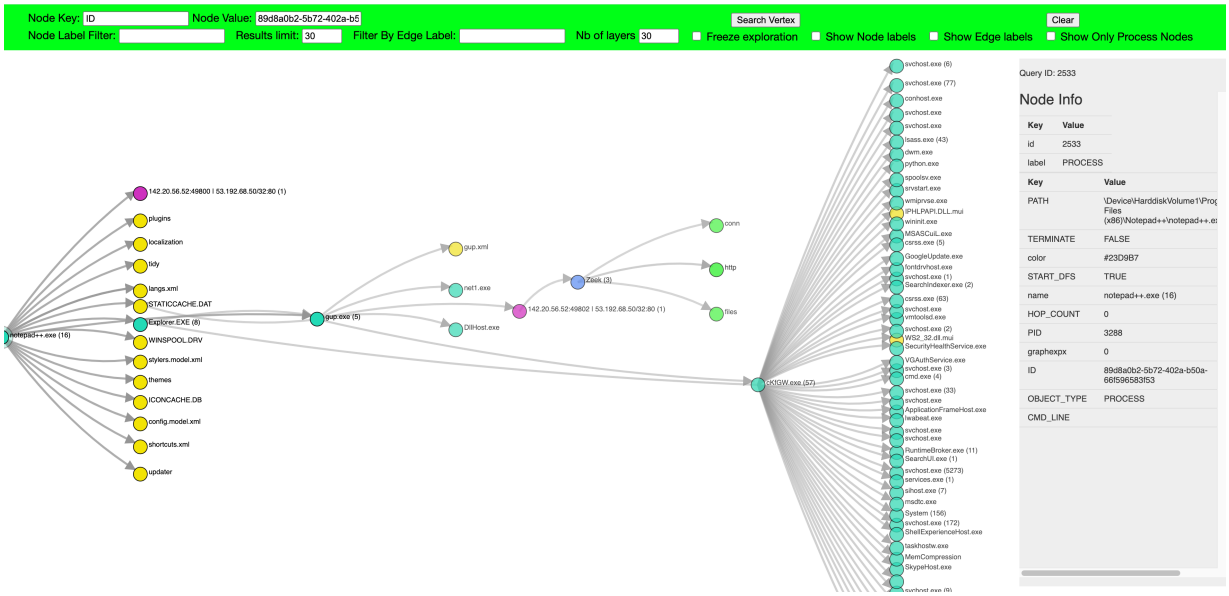


Figure 6.6: Screenshot of our interactive provenance graph visualization tool while investigating Malicious Upgrade attack from DARPA dataset.

attack payload that provides an interactive shell to the attacker from which to explore the target machine and execute code. This binary, `gup.exe` used `CMD` to figure out information about local system and ran ARP scanner on 142.20.56.0/22. After that this binary used Meterpreter "enum_domain" modules to identify domain controller and "enum_shares" to identify any shares on host. After it migrated from process "cKfGW.exe" to "LSASS". The attacker then ran Mimikatz to collect clear-text passwords and established persistence via Meterpreter. Finally, the attacker RDPed to the machine from attacker server.

Investigation The unified provenance graph generated by our visualization tool was able to completely capture this multistage attack. A screenshot of our visualization tool with the unified graph is shown in Figure 6.6. We have shown only part of the graph for readability. The administrator can easily see in this unified provenance graph that how `notepad++` process spawned a malicious-looking process that made a connection to an outside server. The security analysts can further click on the vertex labeled "Zeek" to see other details related to this connection, such as HTTP responses and file hashes, which were impossible to extract using traditional provenance graph techniques.

CHAPTER 7: RELATED WORK

7.1 INTRUSION DETECTION

Existing intrusion 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 [155, 156]. 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.* [157] considered syscall arguments in addition to syscall sequences for malicious program detection. Sekar *et al.* [158] 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 and RapSheet 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 [159, 160]. However, these methods also have significant detection error and suffer from generating too many false alerts [47, 161].

7.2 THREAT ALERT TRIAGE

Ben-Asher *et al.* [42] 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.* [49] mined past analysts' operation traces to generate a state machine for automated alert triage. Chyssler *et al.* [52] 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 [50, 51]; however, there are still too many threat alerts for the analysts to properly investigate [52]. 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.

In the absence of provenance-based causality, alert correlation is another technique to assist analysts by correlating similar alerts. Existing systems use statistical-, heuristic-,

and probabilistic-based alert correlation [162, 163, 164, 165] to correlate alerts. Similar approaches are used in industry for building SIEMs [39, 166]. These techniques are based on feature correlations that do not establish causality. In contrast, RapSheet can establish actual system-layer dependencies between events.

7.3 PROVENANCE ANALYSIS

A lot of work has been done to leverage data provenance for forensic analysis [10, 13, 167, 168], network debugging [26, 27], and access control [28]. Jiang *et al.* [169] used process coloring approach to identify the intrusion entry point and then use taint propagation approach to reduce log entries. Xie *et al.* [170] 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 [20] 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.

Elsewhere in the literature, several provenance-based tools have been proposed for network debugging and troubleshooting [26, 171, 172, 173, 174]. Chen et al. [172] introduced the concept of differential provenance to perform precise root-cause analysis by reasoning about differences between provenance trees. Zeno [171] proposed temporal provenance to diagnose timing-related faults in networked systems. Using sequencing edges Zeno was able to explain why the event occurred at a particular time. RapSheet also uses the sequencing edges but to reason about dependencies between different attack tactics. Zhou et al. [175] designed SNoopy a provenance-based forensic system for distributed systems that can work under adversarial settings.

7.4 LOG REDUCTION

An important component of RapSheet is the log reduction algorithm, which is a topic that is well-studied in recent literature [19, 22, 25]. In the early stages of this dissertation, we realized that existing log reduction techniques were inapplicable to RapSheet design

because they did not preserve the necessary connectivity between EDR generated alerts. For example, LogGC [19] removes *unreachable* events, and thus would not be able to correlate alerts that were related through garbage-collected paths. Similarly, Hossain et al.'s dependence-preserving data compaction technique [25] does not consider that some edges are alert events and must, therefore, be preserved. Similarly, several provenance graph compression techniques [18, 24, 111] are proposed in the literature to reduce the space overhead of provenance collection. Steven et al. [176] proposed a provenance visualization technique which can facilitate investigator in data provenance analysis.

7.5 DISTRIBUTED SYSTEM TRACING

End-to-end tracing is required in distributed systems to enable comprehensive profiling. Existing tools, such as Dtrace [177], Dapper [178], X-trace [179], MagPie [180], Fay [181], and PivotTracing [182] instrument the underlying application to log key metrics at run time. On the other hand, lprof [132] and Stitch [133] allow users to profile a single request without instrumenting any distributed application. lprof uses static analysis to find identifiers that can distinguish output logs of different requests. However, lprof only correlate logs from the same distributed application. On the other hand, Stitch requires certain identifiers in the log messages in order to correlate log messages across different distributed applications. Finally, both systems capture mere correlations instead of true causality between application logs and that can reduce the accuracy of attack reconstruction.

CHAPTER 8: CONCLUSIONS

As the complexity of modern systems continues to increase and attack techniques continue to evolve, I believe that the threat detection and investigation research will only grow in relevance and applicability. This dissertation presents scalable solutions for securing computer systems using principled approaches based on data provenance and program analysis. I showed that data provenance in tandem with program analysis gives security analysts the high ground – a capability that enables them to accurately perform causality analysis, identify the root cause and impact of intrusions, and effectively prioritize and contextualize threat alerts.

NoDoze features contextual information of generated alerts to automatically triage alerts. It uses a novel network diffusion algorithm to propagate anomaly scores in the provenance graphs of alerts and generates aggregate anomaly scores for each alert. NoDoze then uses these aggregate anomaly scores to triage threat alerts. Evaluation results show that NoDoze substantially reduces the slog of investigating false alarms.

RapSheet provides a viable solution for incorporating data provenance into commercial Endpoint Detection and Response (EDR) tools. RapSheet uses the notion of tactical provenance to reason about causally related threat alerts and then encodes those related alerts into a tactical provenance graph. These tactical provenance graphs are later used for triaging EDR-generated threat alerts and for system log reduction.

Provenance analysis can accelerate alert investigation, but only after a provenance graph of the threat alert has been constructed. Unfortunately, before this dissertation, all the provenance graph generation techniques were mainly offline and took hours or days to respond to investigator queries. To address this limitation, I designed the Swift framework. Swift provides high-throughput causality tracking and real-time provenance graph generation capabilities. Swift consists of an in-memory graph database that enables space-efficient graph storage and a hierarchical storage system that keeps the forensically-relevant part of the provenance graph in the main memory while evicting rest to disk. Swift is capable of identifying in-progress threats and provides quick investigation capabilities to a security analyst before serious damage is inflicted.

To solve the semantic gap and dependency explosion problems that currently exist in causality analysis frameworks, I presented OmegaLog. Using static binary analysis techniques, OmegaLog automatically parses dispersed and heterogeneous application event log messages at runtime and associates each record with the appropriate abstractions in the system-level provenance graph. This association allows OmegaLog to transparently solve

both the dependency explosion problem (by identifying event-handling loops through the application event sequences) and the semantic gap problem (by grafting application event logs onto the whole-system provenance graph).

Finally, to further stimulate threat detection and investigation research, we designed an open source and easy-to-deploy endpoint monitoring framework called Zeek Agent. Zeek Agent transparently correlates endpoint events with the Zeek network logs to provide deeper visibility into enterprise-wide activities. Moreover, Zeek Agent allows security analysts to construct and visualize a unified provenance graph from the correlated endpoint and network logs to accelerate the threat investigation process.

REFERENCES

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

[2] "Equifax says cyberattack may have affected 143 million in the U.S." https://www.nytimes.com/2017/09/07/business/equifax-cyberattack.html.

[3] "The SolarWinds Cyber-Attack: What You Need to Know," https://www.cisecurity.org/solarwinds/.

[4] "Cyber Threat Hunting Review," https://blog.usejournal.com/cyber-threat-hunting-basics-52fca11a4e1d, 2019.

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

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

[7] 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.

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

[9] "Splunk," https://www.splunk.com.

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

[11] M. N. Hossain, S. M. Milajerdi, J. Wang, B. Eshete, R. Gjomemo, R. Sekar, S. D. Stoller, and V. Venkatakrishnan, "SLEUTH: Real-time attack scenario reconstruction from COTS audit data," in *USENIX Security Symposium*, 2017.

[12] 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 *Symposium on Network and Distributed System Security (NDSS)*, 2018.

[13] K. H. Lee, X. Zhang, and D. Xu, "High accuracy attack provenance via binary-based execution partition," in *Symposium on Network and Distributed System Security (NDSS)*, 2013.

[14] S. Ma, K. H. Lee, C. H. Kim, J. Rhee, X. Zhang, and D. Xu, "Accurate, low cost and instrumentation-free security audit logging for Windows," in *Annual Computer Security Applications Conference (ACSAC)*, 2015.

[15] 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 *International World Wide Web Conference (WWW)*, 2017.

[16] 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.

[17] S. M. Milajerdi, B. Eshete, R. Gjomemo, and V. Venkatakrishnan, "POIROT: Aligning attack behavior with kernel audit records for cyber threat hunting," in *ACM Conference on Computer and Communications Security (CCS)*, 2019.

[18] W. U. Hassan, M. Lemay, N. Aguse, A. Bates, and T. Moyer, "Towards scalable cluster auditing through grammatical inference over provenance graphs," in *Symposium on Network and Distributed System Security (NDSS)*, 2018.

[19] K. H. Lee, X. Zhang, and D. Xu, "LogGC: Garbage collecting audit log," in *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[20] 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 *Symposium on Network and Distributed System Security (NDSS)*, 2018.

[21] S. Ma, X. Zhang, and D. Xu, "ProTracer: Towards practical provenance tracing by alternating between logging and tainting," in *Symposium on Network and Distributed System Security (NDSS)*, 2016.

[22] 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 *ACM Conference on Computer and Communications Security (CCS)*, 2016.

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

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

[25] M. N. Hossain, J. Wang, R. Sekar, and S. D. Stoller, "Dependence-preserving data compaction for scalable forensic analysis," in *USENIX Security Symposium*, 2018.

[26] 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.

[27] Y. Wu, A. Chen, A. Haeberlen, W. Zhou, and B. T. Loo, "Automated network repair with meta provenance," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[28] J. Park, D. Nguyen, and R. Sandhu, "A provenance-based access control model," in *PST*, 2012, pp. 137–144.

[29] Q. Wang, W. U. Hassan, D. Li, K. Jee, X. Yu, K. Zou, J. Rhee, Z. Chen, W. Cheng, C. A. Gunter et al., "You are what you do: Hunting stealthy malware via data provenance analysis." in *Symposium on Network and Distributed System Security (NDSS)*, 2020.

[30] 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 *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[31] W. U. Hassan, A. Bates, and D. Marino, "Tactical provenance analysis for endpoint detection and response systems," in *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[32] W. U. Hassan, D. Li, K. Jee, X. Yu, K. Zou, D. Wang, Z. Chen, Z. Li, J. Rhee, J. Gui et al., "This is why we can't cache nice things: Lightning-fast threat hunting using suspicion-based hierarchical storage," in *Annual Computer Security Applications Conference (ACSAC)*, 2020.

[33] W. U. Hassan, M. A. Noureddine, P. Datta, and A. Bates, "OmegaLog: High-fidelity attack investigation via transparent multi-layer log analysis," in *Symposium on Network and Distributed System Security (NDSS)*, 2020.

[34] W. U. Hassan, J. Brim, Z. Ansari, M. U. Rehman, M. H. Shahzad, F. Zaffar, and V. Paxson, "Zeek Agent: Correlating host and network logs for better forensics," 2021.

[35] "Zeek: Network Security Monitoring Tool," https://zeek.org/.

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

[37] C. Kruegel, F. Valeur, and G. Vigna, *Intrusion detection and correlation: challenges and solutions*. Springer Science & Business Media, 2004.

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

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

[40] "Netwrix Auditor," https://www.netwrix.com/network_auditing_software_features.html.

[41] 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.

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

[43] S. T. King and P. M. Chen, "Backtracking intrusions," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

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

[45] "200,000+ Systems Affected by WannaCry Ransom Attack," https://www.statista.com/chart/9399/wannacry-cyber-attack-in-numbers/.

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

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

[48] S. Axelsson, "The base-rate fallacy and the difficulty of intrusion detection," *ACM Transactions on Information and System Security (TISSEC)*, 2000.

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

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

[51] A. Laszka, J. Lou, and Y. Vorobeychik, "Multi-defender strategic filtering against spear-phishing attacks." in *AAAI Conference on Artifical Intelligence (AAAI)*, 2016.

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

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

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

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

[56] "The Linux audit daemon," https://linux.die.net/man/8/auditd.

[57] 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.

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

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

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

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

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

[63] "VPNFilter: New Router Malware with Destructive Capabilities," https://symantec-enterprise-blogs.security.com/blogs/threat-intelligence/vpnfilter-iot-malware.

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

[65] 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.

[66] P. Macko and M. Seltzer, "Provenance map orbiter: Interactive exploration of large provenance graphs." in *USENIX Workshop on the Theory and Practice of Provenance (TaPP)*, 2011.

[67] 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.

[68] "Whats in a name? TTPs in Info Sec," https://posts.specterops.io/whats-in-a-name-ttps-in-info-sec-14f24480ddcc.

[69] "The Critical Role of Endpoint Detection and Response," https://bit.ly/39NrNwo.

[70] "MITRE ATT&CK," https://attack.mitre.org.

[71] "Why MITRE ATT&CK Matters," https://symantec-blogs.broadcom.com/blogs/expert-perspectives/why-mitre-attck-matters.

[72] "Experts advocate for ATT&CK," https://www.cyberscoop.com/mitre-attck-framework-experts-advocate/.

[73] "ATT&CK Evaluations," https://attackevals.mitre.org/.

[74] "Endpoint Detection and Response Solutions Market," https://www.gartner.com/reviews/market/endpoint-detection-and-response-solutions.

[75] "File Deletion," https://attack.mitre.org/techniques/T1107/.

[76] "An ESG Research Insights Report," http://pages.siemplify.co/rs/182-SXA-457/images/ESG-Research-Report.pdf.

[77] "About purging reports," https://support.symantec.com/us/en/article.howto129116.html.

[78] "Evaluating Endpoint Products," https://redcanary.com/blog/evaluating-endpoint-products-in-a-crowded-confusing-market/.

[79] "Threat-based Defense," https://www.mitre.org/capabilities/cybersecurity/threat-based-defense.

[80] E. M. Hutchins, M. J. Cloppert, and R. M. Amin, "Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains," *Leading Issues in Information Warfare & Security Research*, vol. 1, no. 1, p. 80, 2011.

[81] "MITRE Matrix," https://attack.mitre.org/matrices/enterprise/.

[82] "APT 29 - Put up your Dukes," https://www.anomali.com/blog/apt-29-put-up-your-dukes.

[83] "APT29," https://attack.mitre.org/groups/G0016/.

[84] "CrowdStrike," https://www.crowdstrike.com/.

[85] "Inside the Cyberattack That Shocked the US Government," ttps://www.wired.com/2016/10/inside-cyberattack-shocked-us-government/.

[86] "Endgame - Endpoint Protection," https://www.endgame.com/sites/default/files/architecturesolutionbrief.pdf.

[87] "Monitoring ALPC Messages," http://blogs.microsoft.co.il/pavely/2017/02/12/monitoring-alpc-messages/.

[88] 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 *Annual Computer Security Applications Conference (ACSAC)*, 2016.

[89] S. M. Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. N. Venkatakrishnan, "HOLMES: Real-time apt detection through correlation of suspicious information flows," in *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[90] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, 1978.

[91] "Common Attack Pattern Enumeration and Classification," https://capec.mitre.org.

[92] "Cyber Threat Intelligence Repository," https://github.com/mitre/cti.

[93] "Registry Run Keys / Startup Folder," https://attack.mitre.org/techniques/T1060/.

[94] "CAPEC-270: Modification of Registry Run Keys," https://capec.mitre.org/data/definitions/163.html.

[95] "Apache TinkerPop," http://tinkerpop.apache.org/.

[96] "RedisGraph - a graph database module for Redis," https://oss.redislabs.com/redisgraph/.

[97] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "MICA: A holistic approach to fast in-memory key-value storage," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[98] "CALDERA," https://www.mitre.org/research/technology-transfer/open-source-software/caldera.

[99] "APT3," https://attack.mitre.org/groups/G0022/.

[100] "Incident investigation," https://www.fireeye.com/solutions/incident-investigation.html.

[101] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver, "Inside the slammer worm," *IEEE Symposium on Security and Privacy (S&P)*, 2003.

[102] "Petya ransomware outbreak," https://www.symantec.com/blogs/threat-intelligence/petya-ransomware-wiper.

[103] "How WannaCrypt attacks," https://www.zdnet.com/article/how-wannacrypt-attacks/.

[104] "Breach Detection," https://link.medium.com/6HpgbLgZuW.

[105] "MTTD vs MTTK," https://www.threatstack.com/blog/how-to-use-automation-to-decrease-mean-time-to-know.

[106] A. Gehani and D. Tariq, "SPADE: Support for provenance auditing in distributed environments," in *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, 2012.

[107] Y. Ji, S. Lee, E. Downing, W. Wang, M. Fazzini, T. Kim, A. Orso, and W. Lee, "Rain: Refinable attack investigation with on-demand inter-process information flow tracking," in *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[108] S. T. King, Z. M. Mao, D. G. Lucchetti, and P. M. Chen, "Enriching intrusion alerts through multi-host causality." in *Symposium on Network and Distributed System Security (NDSS)*, 2005.

[109] M. N. Hossain, S. Sheikhi, and R. Sekar, "Combating dependence explosion in forensic analysis using alternative tag propagation semantics," in *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[110] X. Han, T. Pasqueir, A. Bates, J. Mickens, and M. Seltzer, "Unicorn: Runtime provenance-based detector for advanced persistent threats," in *Symposium on Network and Distributed System Security (NDSS)*, 2020.

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

[112] Y. Xie, K.-K. Muniswamy-Reddy, D. D. E. Long, A. Amer, D. Feng, and Z. Tan, "Compressing provenance graphs," in *USENIX Workshop on the Theory and Practice of Provenance (TaPP)*, 2011.

[113] "Apache Kafka," https://kafka.apache.org/.

[114] C. Kruegel and G. Vigna, "Anomaly detection of web-based attacks," in *ACM Conference on Computer and Communications Security (CCS)*, 2003.

[115] M. Cova, D. Balzarotti, V. Felmetsger, and G. Vigna, "Swaddler: An approach for the anomaly-based detection of state violations in web applications," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2007, pp. 63–86.

[116] C. Kruegel, D. Mutz, W. Robertson, and F. Valeur, "Bayesian event classification for intrusion detection," in *Annual Computer Security Applications Conference (ACSAC)*, 2003.

[117] P. Kumar and H. H. Huang, "GraphOne: A data store for real-time analytics on evolving graphs," in *fast*, 2019.

[118] "Google core libraries for Java," https://github.com/google/guava.

[119] "RocksDB — A persistent key-value store," https://rocksdb.org/.

[120] C. Miller and C. Valasek, "Remote exploitation of an unaltered passenger vehicle," 2015.

[121] "Redis in-memory data structure store," https://redis.io/.

[122] "Over 18,000 Redis Instances Targetted," https://duo.com/decipher/over-18000-redis-instances-targeted-by-fake-ransomware.

[123] Q. Fu, J. Zhu, W. Hu, J.-G. Lou, R. Ding, Q. Lin, D. Zhang, and T. Xie, "Where do developers log? an empirical study on logging practices in industry," in *ICSE Companion*. ACM, 2014.

[124] R. Ding, H. Zhou, J.-G. Lou, H. Zhang, Q. Lin, Q. Fu, D. Zhang, and T. Xie, "Log2: A cost-aware logging mechanism for performance diagnosis," in *USENIX Annual Technical Conference (ATC)*, 2015.

[125] A. Pecchia, M. Cinque, G. Carrozza, and D. Cotroneo, "Industry practices and event logging: Assessment of a critical software development process," in *International Conference on Software Engineering (ICSE)*, 2015.

[126] D. Tariq, M. Ali, and A. Gehani, "Towards automated collection of application-level data provenance," in *USENIX Workshop on the Theory and Practice of Provenance (TaPP)*, 2012.

[127] E. Gessiou, V. Pappas, E. Athanasopoulos, A. D. Keromytis, and S. Ioannidis, "Towards a universal data provenance framework using dynamic instrumentation," in *Information Security and Privacy Research*, D. Gritzalis, S. Furnell, and M. Theoharidou, Eds. Springer Berlin Heidelberg, 2012.

[128] K.-K. Muniswamy-Reddy, U. Braun, D. A. Holland, P. Macko, D. Maclean, D. Margo, M. Seltzer, and R. Smogor, "Layering in provenance systems," in *USENIX Annual Technical Conference (ATC)*, 2009.

[129] Q. Fu, J.-G. Lou, Y. Wang, and J. Li, "Execution anomaly detection in distributed systems through unstructured log analysis," in *ICDM*. IEEE, 2009.

[130] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 2009.

[131] X. Yu, P. Joshi, J. Xu, G. Jin, H. Zhang, and G. Jiang, "Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs," *ACM SIGOPS Operating Systems Review*, vol. 50, no. 2, 2016.

[132] X. Zhao, Y. Zhang, D. Lion, M. F. Ullah, Y. Luo, D. Yuan, and M. Stumm, "lprof: A non-intrusive request flow profiler for distributed systems," in *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2014.

[133] X. Zhao, K. Rodrigues, Y. Luo, D. Yuan, and M. Stumm, "Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle." in *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2016.

[134] A. Oliner, A. Ganapathi, and W. Xu, "Advances and challenges in log analysis," *Communications of the ACM*, vol. 55, no. 2, 2012.

[135] M. Stamatogiannakis, P. Groth, and H. Bos, "Looking inside the black-box: Capturing data provenance using dynamic instrumentation," in *IPAW*. Springer-Verlag New York, Inc., 2015.

[136] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1978.

[137] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive techniques in binary analysis," in *IEEE Symposium on Security and Privacy (S&P)*, 2016.

[138] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, "A large-scale analysis of the security of embedded firmwares," in *USENIX Security Symposium*, 2014.

[139] R. Gerhards, "The syslog protocol," Internet Requests for Comments, RFC 5424, 2009.

[140] "Log4c : Logging for C library," http://log4c.sourceforge.net/.

[141] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, 1976.

[142] "Rsyslogd," http://man7.org/linux/man-pages/man8/rsyslogd.8.html.

[143] "Apache HTTP server benchmarking tool," https://httpd.apache.org/docs/2.4/programs/ab.html.

[144] "Benchmark for ftp servers," https://pypi.python.org/pypi/ftpbench.

[145] "DTrace," https://www.freebsd.org/doc/handbook/dtrace.html.

[146] "Speed considerations," https://github.com/angr/angr-doc/blob/master/docs/speed.md.

[147] S. Haas, R. Sommer, and M. Fischer, "Zeek-Osquery: Host-network correlation for advanced monitoring and intrusion detection," in *ICT Systems Security and Privacy Protection*. Springer International Publishing, 2020.

[148] "Announcing the Zeek Agent," https://zeek.org/2020/03/23/announcing-the-zeek-agent/.

[149] "Performant Endpoint Visibility," https://osquery.io/.

[150] "Endpoint Security Framework," https://developer.apple.com/documentation/endpointsecurity.

[151] "OpenBSM: Open Source Basic Security Module (BSM) Audit Implementation," http://www.trustedbsd.org/openbsm.html.

[152] "DARPA OPTC," https://github.com/FiveDirections/OpTC-data.

[153] K. Cheung and J. Hunter, "Provenance explorer–customized provenance views using semantic inferencing," in *International Semantic Web Conference*. Springer, 2006, pp. 215–227.

[154] T. Kohwalter, T. Oliveira, J. Freire, E. Clua, and L. Murta, "Prov Viewer: A graph-based visualization tool for interactive exploration of provenance data," in *International Provenance and Annotation Workshop*. Springer, 2016, pp. 71–82.

[155] S. A. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion detection using sequences of system calls," *Journal of computer security*, 1998.

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

[157] 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.

[158] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, "A fast automaton-based method for detecting anomalous program behaviors," in *IEEE Symposium on Security and Privacy (S&P)*, 2001.

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

[160] 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.

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

[162] A. Valdes and K. Skinner, "Probabilistic alert correlation," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2001, pp. 54–68.

[163] W. Wang and T. E. Daniels, "A graph based approach toward network forensics analysis," *ACM Transactions on Information and System Security (TISSEC)*, 2008.

[164] H. Debar and A. Wespi, "Aggregation and correlation of intrusion-detection alerts," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2001, pp. 85–103.

[165] Y. Shen and G. Stringhini, "Attack2vec: Leveraging temporal word embeddings to understand the evolution of cyberattacks," in *USENIX Security Symposium*, 2019.

[166] "What is SIEM?" https://logz.io/blog/what-is-siem/.

[167] D. Pohly, S. McLaughlin, P. McDaniel, and K. Butler, "Hi-Fi: Collecting high-fidelity whole-system provenance," in *Annual Computer Security Applications Conference (ACSAC)*, 2012.

[168] Q. Wang, W. U. Hassan, A. Bates, and C. Gunter, "Fear and logging in the internet of things," in *Symposium on Network and Distributed System Security (NDSS)*, 2018.

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

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

[171] Y. Wu, A. Chen, and L. T. X. Phan, "Zeno: Diagnosing performance problems with temporal provenance," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[172] A. Chen, Y. Wu, A. Haeberlen, W. Zhou, and B. T. Loo, "The good, the bad, and the differences: Better network diagnostics with differential provenance," in *ACM SIGCOMM*, 2016.

[173] Y. Wu, M. Zhao, A. Haeberlen, W. Zhou, and B. T. Loo, "Diagnosing missing events in distributed systems with negative provenance," *ACM SIGCOMM Computer Communication Review*, 2014.

[174] W. Zhou, M. Sherr, T. Tao, X. Li, B. T. Loo, and Y. Mao, "Efficient querying and maintenance of network provenance at internet-scale," in *ACM SIGMOD*, 2010.

[175] W. Zhou, Q. Fei, A. Narayan, A. Haeberlen, B. T. Loo, and M. Sherr, "Secure network provenance," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

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

[177] B. Cantrill, M. W. Shapiro, A. H. Leventhal et al., "Dynamic instrumentation of production systems." in *USENIX Annual Technical Conference (ATC)*, 2004.

[178] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," Google, Inc, Tech. Rep., 2010.

[179] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "X-trace: A pervasive network tracing framework," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[180] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using magpie for request extraction and workload modelling." in *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2004.

[181] U. Erlingsson, M. Peinado, S. Peter, M. Budiu, and G. Mainar-Ruiz, "Fay: Extensible distributed tracing from kernels to clusters," *ACM Trans. Comput. Syst.*, vol. 30, 2012.

[182] J. Mace, R. Roelke, and R. Fonseca, "Pivot Tracing: Dynamic causal monitoring for distributed systems," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2015.