Building a Practical Provenance-based Intrusion Detection and Reporting System

by

Jinyuan Liang

B. Eng., Information Security, Central South University, 2019
M. Sc., Cyber Security, University of Birmingham, 2020

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Computer Science)

The University of British Columbia
(Vancouver)

April 2024

© Jinyuan Liang, 2024
The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

**Building a Practical Provenance-based Intrusion Detection and Reporting System**

submitted by Jinyuan Liang in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

**Examining Committee:**

Thomas Pasquier, Assistant Professor, Computer Science, UBC  
*Supervisor*

Margo Seltzer, Professor, Computer Science, UBC  
*Supervisory Committee Member*
Abstract

In computer systems, provenance graphs describe causal relationships among operating system entities (e.g., processes, files, and sockets) to represent a system’s execution history. Provenance-based Intrusion Detection Systems analyze these graphs to identify malicious execution patterns. Despite advances in Provenance-based Intrusion Detection Systems, measurements of detection performance often neglect the quality of detection reports. Prior work either generates coarse-grained alerts or generates fine-grained alerts (e.g., node-level alerts indicating which nodes are suspicious in a graph) with many false positives. This results in security analysts grappling with overwhelming and often irrelevant data, leading to alert fatigue and frequent burnout. To address this issue, we present a node-level detector, PROVNET. Given a provenance graph, PROVNET detects abnormal nodes and generates node-level alerts using a temporal graph autoencoder framework. Subsequently, PROVNET correlates the alerts to mitigate false positives. Based on correlation results, PROVNET then reconstructs the attack subgraphs and generates the detection report to help security analysts investigate the attack execution flow. PROVNET is evaluated against state-of-the-art systems on publicly available datasets, focusing on detection and run-time performance, and robustness. The evaluation results show that PROVNET achieves competitive detection performance compared with other state-of-the-art systems. In addition, the evaluation results demonstrate that PROVNET can perform detection at run-time with low latency, and showcase its robustness against state-of-the-art provenance-based evasion attacks.
Lay Summary

Provenance graphs are like a story of a computer’s activity, showing how different parts (such as files and processes) interact. To spot hacking attempts, Provenance-based Intrusion Detection Systems study these stories for signs of malicious behavior. However, current systems either send out too coarse-grained alarms with insufficient information or too many false alarms at the detailed level, overwhelming security experts with too much, often unhelpful, information. Our solution, PROVNET, intelligently identifies suspicious parts in these graphs and reduces false alarms by correlating alarms. It then creates an insightful report that guides experts through the potential hacking event. Tested on known datasets, PROVNET excels at precisely spotting security breaches and operates swiftly, even against sophisticated hacking strategies designed to evade detection.
Preface

All work presented in the thesis except the design of time window correlation (Section 3.7) is the original work of Jinyuan Liang. The time window correlation approach was designed by Zijun Cheng in the paper where I am a co-author:


The remaining designs, implementation, and experiments were conducted under the supervision of my supervisor Dr. Thomas Pasquier and my project advisor Dr. Xueyuan Han from the Department of Computer Science at Wake Forest University.
Table of Contents

Abstract ................................................................. iii
Lay Summary ............................................................. iv
Preface ................................................................. v
Table of Contents ....................................................... vi
List of Tables .......................................................... ix
List of Figures .......................................................... x
List of Abbreviations .................................................... xiii
Acknowledgments ....................................................... xiv

1 Introduction .......................................................... 1

2 Background & Related Work ......................................... 4
  2.1 Advanced Persistent Threat ....................................... 4
  2.2 System Provenance ............................................... 6
  2.3 Graph Neural Network ........................................... 7
    2.3.1 Definitions ................................................ 7
    2.3.2 Graph Anomaly Detection with Graph Neural Network (GNN) 8
  2.4 Provenance-based Intrusion Detection System ................... 9
  2.5 Report Quality of Intrusion Detection ......................... 10
5.2 Future Work ........................................ 56

6 Conclusion .......................................... 57

Bibliography .......................................... 58
List of Tables

Table 2.1 Investigation effort from detection reports on the THEIA dataset. 11

Table 3.1 System entities and the corresponding attributes considered by PROVNET. 18

Table 3.2 System events considered by PROVNET. 18

Table 4.1 Summary of the experimental datasets. 34

Table 4.2 DARPA data used for training, validation, and test. The bold days are attack days in which both benign and attack activities exist. The remaining days are benign days with only benign activities. 35

Table 4.3 Time-window-level Experimental Results. The bold values show the best results in the datasets. 37

Table 4.4 Node-Level Experimental Results. The bold values show the best results in the datasets. 38

Table 4.5 Comparison of detection performance between the original and ProvNinja testing sets. 51

Table 5.1 Node-Level detection results of PROVNET on DARPA E5 datasets. 55
# List of Figures

| Figure 2.1 | Madniant’s Attack Lifecycle Model | 5 |
| Figure 2.2 | An example of a provenance graph with attacks. The circles are process nodes, rectangles are file nodes, and diamonds are network socket nodes. Red nodes are attack-related. | 6 |
| Figure 3.1 | Overview of PROVNET’s anomaly detection module. PROVNET first constructs provenance graphs from system logs, then performs random walking and applies Natural Language Processing (NLP) techniques to featurize provenance graphs. Given vectorized graphs, PROVNET uses a temporal graph autoencoder to learn graph representations and detect anomalies in graphs. | 15 |
| Figure 3.2 | Overview of PROVNET’s alert correlation and report generation modules. In each time window, cycles, rectangles and diamonds are processes, files and network sockets separately. Red nodes are attack nodes. Purple nodes are false positive nodes. PROVNET performs temporal file removal to remove some false positives in the second and fifth time windows. Then PROVNET applies time window correlation to detect anomalous queues and generate the reports based on detection results. | 16 |
| Figure 3.3 | The examples of original and semantics-enriched random walks. | 21 |
Figure 3.4  Architecture of PROVNET’s Encoder. PROVNET learns graph representations batch by batch. It uses Multi-Layer Perceptron (MLP)s as the message function to generate edges’ messages. Then it extracts and aggregates node messages. PROVNET uses a Gated Recurrent Unit (GRU) [7] model to learn temporal information and a graph transformer [49] to learn structural information.

Figure 3.5  A simple example of the summarization results.

Figure 4.1  Hyper-parameter impacts on time-window-level detection performance. The y-axis is the evaluated metrics. The x-axis is the values of hyper-parameter settings.

Figure 4.2  Hyper-parameter impacts on node-level detection performance. The y-axis is the evaluated metrics. The x-axis is the values of hyper-parameter settings.

Figure 4.3  Hyper-parameter impacts on runtime performance. The y-axis is the average memory usage in megabytes, and the x-axis is the values of hyper-parameter settings.

Figure 4.4  End-to-end run-time performance of PROVNET on the THEIA dataset. The y-axis is the time consumed to detect a time window. The x-axis is the tested time window.

Figure 4.5  The Mimicry Attack on the THEIA dataset. In the graph, circles are process nodes, rectangles are file nodes, and diamonds are network socket nodes. Red nodes are attack-related, green nodes are benign, and orange nodes are benign nodes inserted by the Mimicry Attack.

Figure 4.6  The evaluation results on the evasion graphs generated through Mimicry Attack. The y-axis is the evaluated metrics. The upper x-axis is the number of benign subgraphs inserted; the lower x-axis is the number of benign edges inserted (in millions).
Figure 4.7  *ProvNinja* on the THEIA dataset. The upper graph is the original attack subgraph. The graph at the bottom is the simplified evasion graph generated by *ProvNinja*. In the graphs, circles are process nodes, rectangles are file nodes, and diamonds are network socket nodes. Red nodes are attack-related, green nodes are benign, and orange nodes are benign nodes inserted by the *ProvNinja*. 50

Figure 5.1  The attack subgraphs detected and missed by *PROVNET*. The red nodes are attack-related. The green nodes are benign. . . . 54
# List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>APT</td>
<td>Advanced Persistent Threats</td>
</tr>
<tr>
<td>EDR</td>
<td>Endpoint Detection and Response</td>
</tr>
<tr>
<td>GNN</td>
<td>Graph Neural Network</td>
</tr>
<tr>
<td>GRU</td>
<td>Gated Recurrent Unit</td>
</tr>
<tr>
<td>LOF</td>
<td>Local Outlier Factor</td>
</tr>
<tr>
<td>MLP</td>
<td>Multi-Layer Perceptron</td>
</tr>
<tr>
<td>NLP</td>
<td>Natural Language Processing</td>
</tr>
<tr>
<td>PIDS</td>
<td>Provenance-based Intrusion Detection System</td>
</tr>
<tr>
<td>TGN</td>
<td>Temporal Graph Network</td>
</tr>
<tr>
<td>WL</td>
<td>Weisfeiler-Lehman</td>
</tr>
</tbody>
</table>
Acknowledgments

First, I would like to thank my supervisor Thomas Pasquier. You gave me great help and support during my studies at UBC. You led me into the academic world and always helped me overcome difficulties when I encountered setbacks in provenance research. Here, I would like to offer my most sincere thanks.

Second, I also want to thank my project advisor Xueyuan Han. From the time I entered UBC to my graduation now, you and Thomas have been guiding me in my research on provenance-based intrusion detection and also provided me with many valuable suggestions.

Then, I want to thank my family. While I was studying at UBC, your encouragement, care, and support were the most important reasons why I was able to overcome all kinds of difficulties and complete my studies.

I also appreciate the help from all of you in the Systopia. I am so lucky that I become a member of this big family.

Besides, to all of you who always ordered Uber Eats or went out with me for food hunting this past year, I am glad to have met you all, and I will cherish the time we spent together. Don’t forget to visit my hometown for a food-hunting adventure when you have the time one day.

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC). Nous remercions le Conseil de recherches en sciences naturelles et en g´enie du Canada (CRSNG) de son soutien.
Chapter 1

Introduction

Advanced Persistent Threats (APT) have brought risks to governments, organizations, and enterprises. From the latest APT trends report from the Global Research and Analysis Team at Kaspersky, the APT organizations continue to develop much more complicated toolsets and conduct more long-term threat campaigns, which increases the difficulty of APT intrusion detection [47]. Although there are some Endpoint Detection and Response (EDR) systems adopted to detect APT attacks, they lack the ability to conduct causality analysis on attacks within a long-term APT campaign [34].

To better identify APT intrusions performed on the protected systems, in the past decades, system provenance has been widely used for APT detection (e.g., StreamSpot [37], HOLMES [40], Unicorn [19], ShadeWatcher [58], Kairos [6]). System provenance naturally captures the causal relationships between system objects and the historical information flow. It is generally represented as an attributed and directed graph, where vertices represent system objects (e.g., process, file, socket), edges represent system events between two objects (e.g., fork, read, connect), and edge directions represent the dependency direction of the system events. Typically, attack behaviors are designed to exfiltrate sensitive information or control or compromise the systems, which are different from normal system behaviors, resulting in different provenance graph structures. Based on this insight, a Provenance-based Intrusion Detection System (PIDS) formulates the detection as identifying the abnormal components (e.g., subgraphs, nodes, edges) from normal
provenance graphs.

Prior work [6, 19, 37, 53, 54, 58] mainly focuses on improving the detection performance, but few of them consider the attribution quality of detection reports. Attribution is a crucial aspect of intrusion detection, which requires security experts to review the PIDSes’ outputs closely. Its goal is to quickly identify and tackle the root causes of an intrusion to minimize the impacts of attacks on enterprises’ or organizations’ assets. In general, a detection report contains the abnormal components detected by PIDSes. Some PIDSes may also output additional information to help attack attribution. For example, ThreaTrace [54], a node-level PIDS, outputs all abnormal nodes and their 2-hop ancestors and descendants in the detection reports. At first glance, such neighborhood information can help security analysts better investigate the root cause of abnormal nodes and the negative impact caused by them. However, such forward and backward analysis may suffer from a dependency explosion problem, which can result in a graph with millions of nodes for analysis [26], thereby generating a poor-quality detection report. Therefore, poor-quality detection reports either flood analysts with an excessive, largely irrelevant data volume or fail to accurately provide any useful contextual information. This can lead to a condition known as alert fatigue [22], which significantly increases the risk of analyst burnout [51].

To address these limitations, this thesis presents PROVNET, a PIDS that not only accurately detects abnormal nodes but automatically generates succinct and easy-to-understand detection reports. PROVNET effectively reduces the amount of time that security experts spend investigating intrusions from detection alerts. To do so, PROVNET applies temporal graph learning techniques to identify anomalous nodes. Typically, in a streaming provenance graph, the neighborhood structure of malicious nodes differs from that of benign nodes. Therefore, PROVNET uses a Temporal Graph Network [48] to learn temporal and structural information and alerts anomalous nodes based on a graph autoencoder architecture. Combined with APT attack features (multi-step attacks), PROVNET further applies alert correlation techniques to mitigate false positive alerts and reconstruct attack chains. Afterward, to help security experts investigate intrusions, PROVNET leverages graph summarization techniques to generate and visualize the summary attack graphs in the detection reports.
The contributions are outlined below.

- Systematically conduct evaluations to investigate human efforts to interpret detection reports from state-of-the-art PIDSes for attack attributions.

- To reduce the human effort in investigating attacks from detection reports, we present PROVNET that detects abnormal system behaviors, mitigates false positives by alert correlation, and generates insightful detection reports with the attack chain reconstructed.

- We evaluate PROVNET’s detection performance with state-of-the-art PIDSes on existing benchmark datasets, the run-time performance of PROVNET and the robustness of PROVNET against state-of-the-art provenance-based evasion techniques.

The rest of the thesis is organized as follows: In Chapter 2, we explain the background knowledge necessary for the thesis and discuss the detection report quality of existing PIDSes. In Chapter 3, we formulate the APT detection as anomaly detection, define the threat model, and provide the design details of PROVNET. In Chapter 4, we conduct comprehensive evaluations for PROVNET in terms of detection performance, hyper-parameter impact, run-time performance, and robustness. Chapter 5 discusses the limitations of PROVNET and outlines future research directions. Finally, in Chapter 6, we conclude the thesis.
Chapter 2

Background & Related Work

We start by introducing the attack chains of Advanced Persistent Threats to help readers understand how PROVNET leverages the features of APT attack chains to correlate detection alerts. Since PROVNET is a provenance-based and graph neural network-based intrusion detection and reporting system, in further subsections, system provenance, graph neural network, and provenance-based intrusion detection systems are introduced. Finally, we select some state-of-the-art PIDSes, which are open-source with reproducible evaluations, as motivated systems to discuss their limitations on intrusion detection and reporting.

2.1 Advanced Persistent Threat

APT has become a serious threat to companies, organizations, and governments in the past few decades. Different from traditional cyber attacks, APT adversaries perform multi-stage intrusion (Advanced) to gain access and establish and maintain a foothold in victim machines (Persistent). They leak or tamper with sensitive information (Threat) for some political or financial purposes [9].

The lifecycle of the APT attack is shown in the Figure 2.1, put forward by the Mandiant APT report.

There are several phases in the APT life cycle:

- **Initial Reconnaissance.** This is the first stage of APT attacks, during which attackers gather victims’ information, including host and network systems,
to prepare for further penetrations.

- **Initial Compromise.** In this phase, attackers exploit the vulnerabilities of the services of targeting machines exposed on the internet and try to obtain permission to run a remote shell.

- **Establish Foothold.** After gaining remote shells, attackers attempt to control the target networks and system by, for example, downloading some trojans from a malicious remote host or setting up the backdoors.

- **Privilege Escalation.** To access some sensitive information, attackers require user accounts with higher privileges. Therefore, they continue to exploit some kernel-level or misconfiguration vulnerabilities to create high-privilege user accounts.

- **Internal Reconnaissance.** In this phase, attackers focus on collecting information about the victims and prepare for lateral penetration.

- **Move Laterally, Maintain Presence.** In these two phases, attackers penetrate other local machines and continue to establish footholds for further intrusions.

In Chapter 4 and Chapter 5, we will use some APT datasets published by DARPA Transparent Computing program [28] to evaluate PROVNET and discuss the limitations of PROVNET during APT detection.
2.2 System Provenance

Provenance records the histories of data objects [5]. In the past two decades, provenance techniques have been developed and applied to the security field, from system provenance capturing [13, 45] to provenance-based intrusion detection [6, 20, 21, 25, 37, 39, 40, 54, 58] and forensic analysis [23], especially in APT defense.

Figure 2.2 is an example of a simplified provenance graph from an attack scenario in DARPA TC dataset [28]. When there is no attack, an Nginx process communicates with a benign remote IP, then opens, reads, and sends the contents of the index.html to the benign remote IP. When the attacks occur, the Nginx process receives a malicious request from a malicious remote IP. Then it forks a malicious subprocess to write a trojan file. Later the trojan file is executed and leaks the sensitive information to another remote malicious IP.

![Figure 2.2: An example of a provenance graph with attacks. The circles are process nodes, rectangles are file nodes, and diamonds are network socket nodes. Red nodes are attack-related.](image)

A system provenance graph can be formally defined as $G(t) = \{V(t), E(t)\}$. The $G(t)$ represents the system state of a computer at time $t$. The $V(t)$ represents the nodes in provenance graphs and the types of nodes are process(es), file(s), and socket(s). The $E(t)$ represents all events between system entities at time $t$. Specifically, given two nodes $v_i$ and $v_j$, we define an event occurring at time $t$ as $e_{i,j}(t) = (v_i, Type_{event}, v_j)$.
2.3 Graph Neural Network

2.3.1 Definitions

Inspired by the Convolutional Neural Networks, researchers apply the insights of convolution operation in graph learning and propose the message-passing mechanisms as the core of Graph Neural Network (GNN) to generate node embeddings [17, 31, 52]. Generally, the node embedding on the target node $v_i$ is defined as [55]:

$$h(l) = f_U(l) \sum_{v_j \in \mathcal{N}(v_i)} f_M(h_{i}^{(l-1)}, h_{j}^{(l-1)}, e_{i,j})$$  \hspace{1cm} (2.1)

$h(l)$ is the node $v_i$’s embedding at the $L$-th layer, $f_U$ and $f_M$ are update function and message aggregation function with learnable parameters. $\mathcal{N}(v_i)$ denotes the neighbors of the node $v_i$. The message aggregation function aggregates the neighborhood’s information of $v_i$. If the graph is directed, the aggregation process will follow the directions of the edges. Once the aggregation process is done, the update function will update the node $v_i$ embeddings based on its embeddings at the $(L-1)$-th layer and the aggregated neighborhood information. Note that the number of GNN layers controls the hops of the neighbors that are used for information aggregation.

With different update and aggregation functions, various GNN frameworks have been designed to learn different kinds of graphs. For example, GCN [31] applies convolution-like propagation operations to generate node embeddings for undirected graphs. GraphSage [17] samples and aggregates partial neighbors of a node to learn the representations for large-scale graphs. When the edges are attributed (e.g., each edge has its type), GAT [52] or GraphTransformer [49] leverages attention mechanisms to encode different impacts of neighbors to node embeddings. To further learn the temporal information in a graph, Twitter [48] proposes the Temporal Graph Network (TGN) architecture to encode the historical changes of a node’s neighborhood when generating the embeddings.
2.3.2 Graph Anomaly Detection with GNN

After GNNs encode the structural information into node embeddings, the embeddings can be used for detecting abnormal nodes or edges with a graph autoencoder architecture. An autoencoder is initially designed for dimensionality reduction or feature learning [59]. An autoencoder consists of two parts: an encoder and a decoder. Encoders are used to extract the features of the input vector into a hidden space. Given hidden feature vectors, decoders attempt to reconstruct the original input vectors. The learning goal is to minimize the reconstruction error such that models can learn the features of the inputs.

When adjusting the ways of training and testing, autoencoders can be adopted in anomaly detection [44]. A common approach is to train an autoencoder with sufficient normal data. As the models are well-trained, the autoencoders can well extract features and reconstruct normal data with low error. During testing, the autoencoders have not learned the features of abnormal data and hence reconstruct the data with high error, thereby detecting anomalies.

Similarly, in graph anomaly detection, a graph autoencoder can be used to detect anomalies within graphs. GNN models serve as encoders to generate the node embeddings; the decoders, typically, are Multi-Layer Perceptron (MLP) models to reconstruct the data. The detection tasks can be categorized into node-level detection and edge-level detection [36]. Node-level detection aims to reconstruct the original node feature vectors. The reconstruction error of a node is regarded as the anomaly score of a node. Edge-level detection focuses on predicting the existence of edges or predicting the types of edges. The reconstruction errors are used to measure the accuracy of edge prediction. Normal edges will be predicted with higher accuracy (lower error), while abnormal edges will have lower prediction accuracy (higher error).

PROVNET is built upon the TGN framework and graph autoencoder architecture but introduces further improvements to better learn from provenance graphs and detect attack nodes. More design details will be discussed in Chapter 3.
2.4 Provenance-based Intrusion Detection System

Provenance-based Intrusion Detection Systems detect attack-related information flow within provenance graphs. They can be categorized into two types: signature-based and anomaly-based.

Signature-based PIDSes. Signature-based PIDSes detect attacks based on known attack patterns. For example, HOLMES [40] defines a list of rules to formally describe various APT attack scenarios based on existing APT knowledge bases. According to these rules, HOLMES tries to identify abnormal system behaviors from provenance graphs and map the attack subgraphs to multiple APT scenarios. Apart from rule lists, POIROT [39] leverages graph alignment techniques to detect APT attacks. Similarly, it starts by collecting open-source Cyber Threat Intelligence reports. Then it constructs attack graphs through descriptions in the reports. During detection, POIROT checks if any subgraphs can be aligned to the attack graphs. The benefit of signature-based approaches is that they can generate easily understandable detection reports with specific attack subgraphs identified. Security analysts can quickly investigate the attacks and conduct appropriate protective measures to minimize the damage from attacks. However, such detection approaches rely on the completeness of attack knowledge bases. They can only detect attacks that are publicly known, whereas APT adversaries often exploit 0-day vulnerabilities and perform complex attacks that have not been shown before to evade detection. This causes the signature-based approaches to fail to detect some sophisticated APT attacks. So, in further discussion or design, we do not consider signature-based approaches.

Anomaly-based PIDSes. The main idea of anomaly-based approaches is to figure out any abnormal provenance data from normal data. In general, whatever kinds of APT attacks are performed, attack behaviors are different from normal system behaviors, resulting in abnormal subgraph structure. Therefore, anomaly-based approaches learn the graph patterns from normal data. During detection, any data that has unobserved patterns will be flagged as anomalies. Since this type of approach does not rely on any attack information, it is widely applied to detect unknown and stealthy attacks in practice [20, 53, 58].

Regardless of its powerful detection ability, anomaly-based approaches suf-
fer from producing too many false positive alerts, because any system behavior different from normal behavior will be alerted, but not all abnormal data is attack-related [2]. Therefore, as will be discussed in Chapter 3, two of design requirements are: 1) PROVNET should be anomaly-based to detect unknown attacks, and 2) PROVNET should not produce too many false positives.

### 2.5 Report Quality of Intrusion Detection

We systematically discuss the limitations of state-of-the-art PIDSes in reporting anomalies to motivate this thesis project through quantitative and design analysis.

#### 2.5.1 Quantitative Analysis

For quantitative analysis, we select a dataset called THEIA published by DARPA Transparent Computing Engagement 3 program [28], which is widely used for benchmarking PIDSes. We select two state-of-the-art node-level detectors, Kairos [6] and ThreaTrace [54], that are open-source and reproduce their evaluations on the THEIA dataset. Then we analyze how much human effort is required by security analysts to interpret the detection report. Though they are evaluated to motivate our project, I also attach the results of our system on the same dataset for better comparison.

In the THEIA dataset, there are 443,978 nodes, where 32 are attack nodes, and they account for only 0.0072% of all nodes. There is only one attack subgraph within the dataset.

Table 2.1 shows a comparison of investigation efforts among Kairos, ThreaTrace, and PROVNET on the THEIA dataset. According to the data presented in the table, Kairos produces a total of 17 subgraphs, with 3 of these containing a mix of attack and benign (false positive) nodes. Kairos uses a community discovery algorithm [3] to generate compact attack subgraphs from the detection outputs. However, the false positive nodes lead to inaccurate classification in Kairos’ community discovery algorithm, resulting in the attack behaviors across three distinct subgraphs.

For ThreaTrace, there is only a large provenance graph generated for attack investigation. After manually checking the graph, there are only 7 nodes are attack-
related, which means the remaining 7,262 nodes are false positives.

By comparison, PROVNET precisely identifies and generates the sole attack subgraph, achieving this without incorporating any false positives (all 25 nodes are attack-related). This level of accuracy substantially alleviates the workload associated with scrutinizing attacks.

<table>
<thead>
<tr>
<th>System</th>
<th># of subgraphs generated for attack investigation</th>
<th># of generated subgraphs containing attacks (True Positives)</th>
<th>Avg # of nodes per subgraph</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROVNET</td>
<td>1</td>
<td>1</td>
<td>25</td>
</tr>
<tr>
<td>Kairos</td>
<td>17</td>
<td>3</td>
<td>630</td>
</tr>
<tr>
<td>ThreaTrace</td>
<td>1</td>
<td>1</td>
<td>7,269</td>
</tr>
</tbody>
</table>

Table 2.1: Investigation effort from detection reports on the THEIA dataset.

2.5.2 Design Analysis

For design analysis, we comprehensively analyze the design limitations of the state-of-the-art systems.

**Unicorn.** Han et al. [19] proposed Unicorn to detect malicious graphs at runtime. Unicorn adapts the Weisfeiler-Lehman (WL) graph kernel algorithm to embed the contextual information of nodes and generate the evolving graph embeddings at different timestamps. Unicorn then learns an evolving model to detect and report abnormal graphs. The limitation of Unicorn is that it performs detection at the graph level. Security analysts can only inspect the abnormality of the graphs from detection reports. However, for further forensic analysis, analysts have to investigate the whole graph to manually figure out the malicious nodes and edges. Such graph-level PIDSes cannot provide much insightful information in the detection report.

**SIGL.** To further reduce the gap between intrusion detection and investigation, Han et al. [20] designed SIGL, a node-level PIDS using graph autoencoder techniques to detect abnormal nodes. SIGL leverages a graph LSTM to embed structural information into node embeddings and uses a 2-layer MLP to reconstruct node feature vectors. Different from Unicorn, SIGL alerts all abnormal nodes in detection reports, helping analyze attacks. However, SIGL does not have any mechanism to mitigate the false positives. Our evaluations in Chapter 4 show that, in the worst
case, SIGL generates 2336 false positives out of 2338 alerts in total, which makes it hard for analysts to investigate attacks.

**ThreaTrace.** To improve detection performance, ThreaTrace [54] uses multiple sub-models to comprehensively learn graph information and perform node-level detection. Such a multi-model framework makes ThreaTrace better capture contextual information for each node, which improves detection performance compared with some single-model detectors. However, provenance graphs are evolving as the system runs, but ThreaTrace does not learn any temporal information from the graphs, still causing lots of false positives (the quantitative analysis above has already shown this, with 7262 false positives out of 7269 alerts in total). In addition, ThreaTrace reports 2-hop neighbors of abnormal nodes. Security analysts have to spend extra effort to investigate all those neighbors to avoid any miss of potential risks, increasing the workload of analysts.

**Kairos.** Kairos [6] is the first PIDS that incorporates run-time anomaly detection, alert correlation, and graph summarization techniques. It featurizes nodes with hierarchical feature hashing techniques [60], learns structural and temporal information using TGN [48], then uses a 2-layer MLP for a link type prediction task to identify abnormal edges and nodes per time window. Kairos assumes that all attack nodes are rare compared to other benign nodes in history. So it leverages a rareness-based approach [8] to filter some common nodes and generate attack chains by correlating abnormal time windows. When generating the detection reports, Kairos also utilizes community discovery techniques [3] to construct multiple attack graphs with different user behaviors. Such user-friendly designs help security analysts quickly pinpoint attacks, but there are still some drawbacks that incur false positives:

1. Provenance graphs reflect complex execution states of a system. Edges vary between two nodes at different time. A simple 2-layer MLP decoder cannot perfectly predict edge types in those complex execution states.

2. When filtering false positives, Kairos only keeps rare nodes and removes common nodes, but not all rare nodes are attack-related. For example, temporary files are rare in provenance graphs, but they do not contribute to attacks [33].
These limitations cause some false positive nodes to remain in detection reports. PROVNET follows the philosophy of Kairos’ designs, leveraging an autoencoder framework to detect anomalies and alert correlation techniques to mitigate false positives.
Chapter 3

Design

3.1 Problem Definition.

We formalize APT intrusion detection as an anomaly detection problem on a series of time-window graphs. The goal is to identify nodes related to malicious system activity performed by an attacker. More specifically, given a series of time-window graphs $G_{TW} = \{G_{TW_1}, G_{TW_2}, \ldots, G_{TW_n}\}$, PROVNET applies feature engineering to vectorize the graphs into $VecG_{TW} = \{VecG_{TW_1}, VecG_{TW_2}, \ldots, VecG_{TW_n}\}$. PROVNET learns detection models to detect abnormal nodes $V_{abnormal}^{TW_i}$ within each time-window graph. After abnormal nodes are alerted, PROVNET should have comprehensive mechanisms to filter false positives and correlate alerts. Finally, based on correlation results, PROVNET generates an attack graph $G_{attack}$ including each APT step in the detection reports.

3.2 Threat Model

We follow a threat model similar to prior work [6, 19, 39, 40, 53]. We assume the attackers follow the steps of APT attack chains to penetrate, control the victim systems, and maintain their presence in the system. Side-channel attacks are orthogonal to this project as their attack behaviors cannot be explicitly captured by kernel-level audit systems.

Additionally, PROVNET requires training with benign data and performing de-
tection with the well-trained model. So we assume the training data does not include any abnormal system behaviors and the attackers cannot temper with the model parameters (i.e., data and model poisoning attacks are out of the scope of our threat model). The entire PROVNET framework, provenance capturing system, and all underlying kernel and hardware components are our trusted computing base. We leverage tamper-evident logging techniques [42, 43] to ensure log integrity and detect any attempt to tamper with provenance logs.

To make PROVNET more practical, we assume the attackers have knowledge of the logging mechanism that underlies the provenance capture infrastructure, design details of PROVNET, and normal execution behaviors of the protected systems. This allows them to use sophisticated state-of-the-art provenance-based evasion attacks [15, 41] to avoid detection.

### 3.3 Workflow of PROVNET

**Figure 3.1:** Overview of PROVNET’s anomaly detection module. PROVNET first constructs provenance graphs from system logs, then performs random walking and applies Natural Language Processing (NLP) techniques to featurize provenance graphs. Given vectorized graphs, PROVNET uses a temporal graph autoencoder to learn graph representations and detect anomalies in graphs.

Figure 3.1 and Figure 3.2 depict the workflow of PROVNET. There are five major parts in PROVNET:

- **Provenance Graph Construction (Section 3.4).** The underlying system logging
Figure 3.2: Overview of PROVNET’s alert correlation and report generation modules. In each time window, cycles, rectangles and diamonds are processes, files and network sockets separately. Red nodes are attack nodes. Purple nodes are false positive nodes. PROVNET performs temporal file removal to remove some false positives in the second and fifth time windows. Then PROVNET applies time window correlation to detect anomalous queues and generate the reports based on detection results.
infrastructure keeps recording system events. PROVNET chronologically parses system events into time-window provenance graphs.

Provenance Graph Featurization (Section 3.5). In a provenance graph, the semantics of a node depend on ancestors and descendants. To featurize the semantics of a node, PROVNET performs random walks to extract contextual information of each node. Then PROVNET applies NLP techniques to vectorize nodes. PROVNET further encodes edges based on their edge types and source and destination nodes.

Anomaly Detection (Section 3.6). PROVNET leverages a graph autoencoder framework for anomaly detection. It uses a Temporal Graph Network (TGN) as the encoder to aggregate both temporal and structural information of the graphs into node embeddings. The decoding process involves using MLPs to reconstruct the feature vectors of the nodes. In the training phase, similar to other node-level tasks, the learning goal is minimizing reconstruction errors. Before the system is deployed, PROVNET uses benign data to determine the threshold for detection. Upon deployment, PROVNET first computes the anomaly scores of edges, based on the reconstruction results of their source and destination nodes. Any edges with anomaly scores higher than the threshold are flagged as abnormal. Their source and destination nodes are hence alerted.

Alert Correlation (Section 3.7). Given abnormal nodes alerted in each time window, PROVNET leverages a density-based outlier detection approach [4] to remove the impact of temporary files as they generally do not contribute to any attack in provenance analysis [33]. Afterward, PROVNET correlates the abnormal time windows by checking if the two abnormal time windows have overlapping abnormal nodes. All abnormal time windows will be correlated into queues. Their anomaly scores are computed based on the anomaly scores of all abnormal edges. If the anomaly score of a queue exceeds the pre-set threshold, all abnormal time windows and nodes will be reported.

Report generation (Section 3.8). After alert correlation, all attack behaviors across abnormal time windows are naturally constructed into an APT attack chain. To help security analysts better understand the attack information flow from detection reports, PROVNET visualizes the attack chain based on the detection results (all abnormal time windows, abnormal nodes, and edges). During visualization, PROVNET applies a graph reduction technique [57] to reduce to size of the attack.
3.4 Provenance Graph Construction

PROVNET processes the system audit logs from Window ETW, Linux Audit, or CamFlow. We follow the same system entities and attributes and event types described in prior work [6] to construct provenance graphs. PROVNET parses three types of system entities and nine types of events into provenance graphs. Table 3.1 shows the system entities and the corresponding attributes considered by PROVNET. Table 3.2 shows the system events considered by PROVNET. Specifically, when constructing provenance graphs, we parse system events into directed edges with timestamps in a streaming fashion. Note that the directions of edges reflect the dependency relationships between system entities, we follow the direction according to prior work [30].

<table>
<thead>
<tr>
<th>System Entities</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process</td>
<td>Executable Name</td>
</tr>
<tr>
<td>File</td>
<td>File Name</td>
</tr>
<tr>
<td>Network Socket</td>
<td>Local IP, Local Port, Remote IP, Remote Port</td>
</tr>
</tbody>
</table>

Table 3.1: System entities and the corresponding attributes considered by PROVNET.

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Source Node Type</th>
<th>Destination Node Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>clone</td>
<td>Process</td>
<td>Process</td>
</tr>
<tr>
<td>exec</td>
<td>File</td>
<td>Process</td>
</tr>
<tr>
<td>open</td>
<td>File</td>
<td>Process</td>
</tr>
<tr>
<td>read</td>
<td>File</td>
<td>Process</td>
</tr>
<tr>
<td>write</td>
<td>Process</td>
<td>File</td>
</tr>
<tr>
<td>close</td>
<td>Process</td>
<td>File</td>
</tr>
<tr>
<td>connect</td>
<td>Process</td>
<td>Network Socket</td>
</tr>
<tr>
<td>send</td>
<td>Process</td>
<td>Network Socket</td>
</tr>
<tr>
<td>receive</td>
<td>Network Socket</td>
<td>Process</td>
</tr>
</tbody>
</table>

Table 3.2: System events considered by PROVNET.
3.5 Provenance Graph Featurization

After provenance graphs are constructed, PROVNET featurizes the nodes and edges such that the graph autoencoder can learn the representation of the graphs.

3.5.1 Node Featurization

A good featurization approach must satisfy the following criteria:

1 **Structural wise.** The feature vector of a node should encode the structural information of a graph. More concretely, if two nodes have the same predecessors and successors, the roles of these two nodes are structurally equivalent, and their feature vectors should be close to each other. For example, for the traces \{\textit{firefox}, write, /tmp/fileA, read, \textit{bash}\} and \{\textit{firefox}, write, /tmp/fileB, read, \textit{bash}\}, both files have the same predecessors and successors. From the perspective of provenance graph structure, these two file nodes have the equivalent role. Their feature vectors should be close to each other in the feature space.

2 **Generalizability.** PROVNET should work inductively. During system execution, it is common that system entities have not been shown in the training data (e.g., temporary files or some IP addresses). If the featurization approach cannot generate the node feature vectors for all kinds of nodes, PROVNET cannot proceed to further graph learning and detection steps. Additionally and importantly, the newly generated feature vectors should be in the same feature space as existing vectors. Otherwise, processing newly generated vectors with existing vectors is meaningless.

**Word2Vec.** To satisfy the first criteria, we apply word2vec [38] to learn the node feature vectors. Word2vec learns high-quality word embeddings by predicting context words for a given target word. This approach effectively captures the syntactic and semantic relationships between words by analyzing their co-occurrence patterns within a large text corpus. More concretely, each word in the corpus is treated as a target word, and the model attempts to predict the words that appear in its surrounding context within a sliding window. The objective is to adjust the word embeddings in a way that maximizes the likelihood of the actual context words...
given the target word. This training process encourages the model to place words with similar contexts close together in the embedding space.

We collect a corpus for word2vec through the random walking algorithm. Generally, the random walking algorithm starts from a node, randomly traverses to one of the next successors through a random edge, and repeats the same process until the traversal length reaches a pre-set length [20]. The algorithm will run multiple times for each node to ensure collecting enough context for a node. The attributes of all traversed nodes and the types of edges will be concatenated into a sentence. The random walk algorithm will run on all nodes to collect the contexts for all nodes.

When word2vec is trained with such sentences, the sliding window mechanism results in that the generation of a node’s embedding is influenced by all its ancestors’ embeddings along the sampled provenance paths. Take the provenance graph in Chapter 2 as an example. Figure 3.3 shows the random walk and the sentence sampled. When word2vec learns from Sentence 1, the sliding window mechanism results in that the embedding of Trojan File is influenced by all embeddings of the ancestors (Malicious IP2 and Nginx).

However, when performing causality analysis for a node, we consider both the ancestors’ and descendants’ information to determine the role of this node. So, we should also encode such information when generating a node’s feature vector. To do so, we design a semantics-enriched random walks approach by improving the original random walking algorithm. We add another sampling process that traverses the graph following the inversed directions of the edges.

The bottom of the Figure 3.3 shows the exampled walks and sentences for the semantics-enriched random walk approach. When generating the embedding of Trojan File, the embedding can encode the ancestors’ information from Sentence 1 and descendants’ information from Sentence 2.

**A La Carte Embedding.** As word2vec is trained with the collected corpus, it can generate high-quality node feature vectors. However, word2vec cannot handle unseen nodes. Therefore, PROVNET adopts the A La Carte embedding model to address the out-of-vocabulary issue [29].

The A La Carte embedding method represents the embedding of a target word $w$ as the product of a learned linear transformation matrix $A$ and the averaged con-
Figure 3.3: The examples of original and semantics-enriched random walks.

text embeddings $v_w^{\text{additive}}$. This matrix $A$ is obtained through a linear regression approach designed to approximate the word embeddings from word2vec by leveraging contextual information. Formally, this is represented by the equation:

$$v_w \approx A v_w^{\text{additive}} = A \left( \frac{1}{C_w} \sum_{c \in C_w} \sum_{w' \in c} v_{w'} \right)$$  \hspace{1cm} (3.1)

Here, $v_w$ is the embedding for the target word, $C_w$ denotes the set of contexts.
of \( w, w' \) are the words within those contexts, and \( v_{w'} \) are their corresponding embeddings. The context embeddings \( v_w^{\text{additive}} \) are computed as the mean of the embeddings of the words in the context. Once learned, the matrix \( A \) can be applied to infer the words that do not exist in the training corpus of the word2vec. Given an unseen word \( f \), its word embedding \( v_f \) is computed through the equation:

\[
v_f \approx A v_f^{\text{additive}} = A \left( \frac{1}{C_f} \sum_{c \in C_f} \sum_{w' \in c} v_{w'} \right)
\]  

(3.2)

By applying semantics-enriched random walking, word2vec, and A La Carte embedding, PROVNET can generate feature vectors for all nodes in provenance graphs.

### 3.5.2 Edge Featurization

We chronologically featurize the edges based on their edge types, source, and destination nodes. The feature vector of an edge is the concatenation of its source node’s feature vector, one-hot encoding of its edge type, and its destination node’s feature vector. Formally, the feature vector of an edge is denoted as \( \text{Vec}_e(t) = \text{CONCAT}(\text{Vec}_{\text{src}}, \text{One} - \text{hot}_{\text{Etype}}, \text{Vec}_{\text{dst}}) \), where the \( \text{Vec}_e(t) \) is the edge feature vector at time \( t \), \( \text{Vec}_{\text{src}} \) and \( \text{Vec}_{\text{dst}} \) are feature vectors of source and destination nodes respectively, and \( \text{One} - \text{hot}_{\text{Etype}} \) is the one-hot encoding of the edge type.

### 3.6 Temporal Graph Learning and Anomaly Detection

#### 3.6.1 Encoder

PROVNET leverages a graph autoencoder to learn the graph representation. A high-quality graph representation should accurately encode both structural and temporal information of the graphs. Most GNNs aggregate neighborhood information to embed a node. However, since a provenance graph evolves as time goes by, we expect the GNN encoder can learn the representation of a dynamic graph chronologically. Given such consideration, we select the TGN model [48] as the graph encoder.

Figure 3.4 shows the framework of the PROVNET’s TGN encoder. TGN satisfies...
Figure 3.4: Architecture of PROVNET’s Encoder. PROVNET learns graph representations batch by batch. It uses MLPs as the message function to generate edges’ messages. Then it extracts and aggregates node messages. PROVNET uses a Gated Recurrent Unit (GRU) [7] model to learn temporal information and a graph transformer [49] to learn structural information.

the requirements that it is designed to learn dynamic graph-structured data, where the interactions between entities are not only rich in structure but also evolve over time. A TGN adeptly captures these temporal interactions by maintaining stateful memory for each node, which is updated as new events occur. It consists of
four parts: Message Function, Message Aggregator, Memory Updater, and a GNN model, where the first three learn the temporal information and the GNN model learns the structural information.

**Message Function.** Each time a new edge occurs in the graph, a TGN uses two different MLPs to generate the node messages by combining the source node $i$, destination node $j$, and time $t$ information. They are computed via:

$$m_i(t) = \text{MLP}_s(s_i(t^-), \Delta t, e_{ij}(t)),$$
$$m_j(t) = \text{MLP}_d(s_j(t^-), \Delta t, e_{ij}(t)) \quad (3.3)$$

Here, $m_i(t)$ and $m_j(t)$ are the message generated for node $i$ and $j$ at time $t$. $s_i(t^-)$ and $s_j(t^-)$ stand for the historical states of the node $i$ and $j$ before time $t$. If a node is newly added to the graph, the state of this node is initialized with zeroes because there is no history information for this node. If the node exists in the past, the state of this node is the result of Memory Updater, which encodes historical state changes of a node from past events. $\Delta t$ denotes the time difference since the last recorded edge. $e_{ij}(t)$ is the edge feature vector. For simplicity, TGN concatenates the $(s_i(t^-), \Delta t, e_{ij}(t))$ as the input of MLP.

**Message Aggregator.** Considering that existing GNNs are trained batch by batch to learn a large-scale graph [17], a node $i$ may be involved in multiple interactions and therefore have different messages $m_i(t)$ at different times. Thus the TGN applies a message aggregator function to aggregate all messages of a node within a training batch. Assuming that $t_1, t_2, \ldots, t_b$ are the time of events in a batch:

$$\hat{m}_i(t) = \text{agg}(m_i(t_1), \ldots, m_i(t_b)) \quad (3.4)$$

The aggregation function can be customized based on the needs of users. Here we adopt the same choice as the TGN paper [48], which is the most recent message (keep only the most recent message for a given node). This may not be as good as some time series learning models such as Long-Short Term Memory [24], but most recent message is a non-learnable approach, which means TGN consumes fewer computational resources and is more practical. In addition, from the evaluation results in Chapter 4, most recent message is enough to achieve good detection performance.
**Memory Updater.** As described above, $s_i(t^-)$ denotes the information of the evolving neighborhood of the node $i$. So the TGN uses it and the current message $\hat{m}_i(t)$ to update its state via:

$$s_i(t) = \text{mem}(\hat{m}_i(t), s_i(t^-))$$  \hspace{1cm} (3.5)

The TGN leverages a GRU model [7] as the memory updater.

**GNN model.** Given temporal information learned by the updater, a GraphTransformer [49] model is applied to embed the structural information into node embeddings.

So far, the information-rich node embeddings have been generated for further anomaly detection.

### 3.6.2 Decoder

The anomaly detection objectives guide the decoder’s design and the corresponding loss function. Typically, security analysts audit events from system logs, identify abnormal operations, and further investigate the corresponding system entities. Therefore, **PROVNET** detects abnormal edges first and then identifies abnormal nodes.

A common edge-level task is link prediction. Given two node embeddings, it uses a single MLP to predict the probability of the existence of an edge between two nodes [32]. However, such a decoder framework is insufficient for provenance edge prediction. On one hand, provenance graph structure reflects the execution pattern of a system, which is much more complicated than other types of graphs (e.g., citation graphs) to predict the existence of edges. On the other hand, the edge types should also be considered for prediction, because edge types vary between two nodes. While combining link existence prediction and link type prediction may obtain good prediction performance on provenance graphs, it is non-trivial to design a decoder framework and related loss functions to achieve this goal.

To devise a decoder that is both structurally simple and efficacious, we reflect on the analytical process of a security expert assessing whether an event is abnormal. A system event is considered abnormal depending on its subject and object. For example, if an unknown process from the /tmp/ folder keeps sending large
amounts of data to a specific remote host, this should be considered abnormal due to the abnormal process (subject) and the abnormal remote host IP (object). From this perspective, the decoder of PROVNET employs two MLPs, one of which is to reconstruct the source node, and the other reconstructs the destination node. The combined reconstruction losses from these MLPs serve as the anomaly score for an edge $\text{AS}(e_{ij})$, which is computed via:

$$\text{AS}(e_{ij}) = \text{Loss}(v_i) + \text{Loss}(v_j)$$  \hspace{1cm} (3.6)

The learning goal of graph autoencoder is to minimize the anomaly score of edges.

Intuitively, when an edge is abnormal, its source and destination nodes are abnormal and have high reconstruction losses. The addition operation further amplifies the anomaly score of the edge, thereby accentuating the distinction between the scores of benign and abnormal edges.

For the loss function, unlike existing PIDSes (e.g., SIGL) that uses Mean Square Error, PROVNET uses the Scaled Cosine Error proposed by Hou et al [27] to measure the reconstruction error. Scaled Cosine Error outperforms Mean Square Error for node reconstruction tasks when the node feature vectors vary in their magnitudes, which is often the case for node attributes in provenance graphs [27]. The Scaled Cosine Error $L_{\text{SCE}}$ is defined as:

$$L_{\text{SCE}} = \frac{1}{|V'|} \sum_{v_i \in V'} \left(1 - \frac{x_i^T z_i}{\|x_i\| \cdot \|z_i\|}\right)^y, \quad y \geq 1,$$  \hspace{1cm} (3.7)

Here, $x_i$ and $z_i$ represent the original feature vectors and the reconstructed feature vectors respectively. $y$ is an adjustable scaling factor that controls the scaling degree. PROVNET follows the setting ($y = 3$) suggested by Hou et al [27] for model training and testing. $V'$ is a list of nodes that are reconstructed.

### 3.6.3 Anomaly Detection

The graph autoencoder models are trained with benign-only data offline and detect anomalies in time windows online. The threshold is set as the highest anomaly score in the validation set. During detection, edges with anomaly scores higher than the threshold are flagged as abnormal. Once abnormal edges are detected,
PROVNET alerts the time windows that have abnormal edges, the abnormal edges, and their source and destination nodes.

### 3.7 Alert Correlation

PROVNET mitigates the false positive alerts in two levels:

1. **Node level.** PROVNET removes false positives caused by temporary files. Note that the temporary files in provenance analysis have different definitions from those in APT attacks. More details are discussed in Section 3.7.1.

2. **Time window level.** PROVNET correlates the abnormal time windows into abnormal queues to remove the false positive time windows.

After alert correlation, PROVNET generates the detection report with attack chains reconstructed.

#### 3.7.1 Temporary File Removal

Many APT attacks often write malicious scripts in temporary files and execute them to perform malicious behaviors. Such kind of temporary files interact with multiple processes during their life cycle. However, in the field of provenance analysis, temporary files are file nodes that only interact with one process during its life cycle, so they do not have any contribution to attacks and are regarded as noises [33].

Note that PROVNET performs detection in a streaming fashion and cannot foresee if a file is a temporary file or not. We cannot simply remove all file nodes with no descendants because they may interact with other processes in the future. In this case, they should be regarded as temporary files. Thus, we do not remove temporary files during the provenance graph construction process. Instead, we remove the false positive alerts containing temporary file nodes. We notice that temporary files interacting with the same process have similar roles in the provenance graphs. Thus their node feature vectors are closed with each other. So if we have a list of temporary files, we can check if a file is a temporary file by measuring the distance between its feature vector and the vectors of existing files.

**Problem definition.** Based on this insight, we formulate a temporary file removal problem as an outlier detection problem. Given the feature vectors of a list of files
vec\(F = vec\(f_1, vec\(f_2, ..., vec\(f_n\)\) and a file to be tested \(f_{\text{test}}\). PROVNET aims to detect if 
vec\(f_{\text{test}}\) is outlier or not compared with \(vec\(F\). If it is an outlier, \(f_{\text{test}}\) does not serve
the same role as existing files and hence is not a temporary file.

To do so, PROVNET follows the steps below for temporary file detection.

Data Collection. PROVNET collects a list of filenames collected in benign data
(in our evaluation, we collect from the training and validation sets of the graph
autoencoder). Then PROVNET vectorizes the filenames using the word2vec model
that is well-trained in the provenance graph featurization process.

Model Training. PROVNET uses Local Outlier Factor (LOF) [4] for outlier de-
tection. LOF is an algorithm used to identify outlier data points by measuring the
local deviation of a given data point with respect to its neighbors. It is based on the
premise that outliers are data points that are distant from the main concentration
of data. The LOF score of a point is calculated by comparing the local density of
a point with the densities of its neighbors. Points that have a substantially lower
density than their neighbors are considered outliers. Given the vectors of the files,
PROVNET trains a LOF model.

Outlier Detection. For the alerts from the autoencoder, PROVNET tests all files
and assigns outlier scores for them. The outlier score \(OS(f_{\text{test}})\) is computed by the equation:

\[
OS(f_{\text{test}}) = (¬\text{EXIST}(f_{\text{test}}, F)) \ast \text{LOF}(vec\(f_{\text{test}}, vec\(F\)) \quad (3.8)
\]

Here, \(F\) is the list of files. \(\text{EXIST()}\) is a filter function that checks if \(f_{\text{test}}\) is in \(F\). If
\(f_{\text{test}}\) is in \(F\), \(\text{EXIST()}\) returns 1; otherwise it returns 0. \(\text{LOF()}\) is the function that
computes the LOF score.

There are two cases when computing the outlier scores: 1) if the tested files
are in the benign data, they are not temporary files and will be assigned 0 after the
negation operation \((¬\text{EXIST}(f_{\text{test}}, F))\); 2) if the tested files are not in the benign
data, the outlier scores of them are the LOF score. To detect if they are temporary
files, we adopt the same threshold as that in the LOF paper [4].

Finally, for all temporary files detected, PROVNET removes them from the
alerts. If a time window does not contain alerts after this step, this time window
will not be correlated in the alert correlation phase as this is a false positive time
window.
3.7.2 Alert Correlation

PROVNET aims to detects APT attacks in provnance graphs. In general, the attackers lurk in the targets for a long time. They perform malicious behaviors across multiple time windows, forming an attack chain [40]. Following such characteristics of APT attacks, PROVNET attempts to correlate abnormal time windows and reconstruct the attack chains. To do so, PROVNET maintains anomalous queues at run-time. For each newly incoming time window, if it is detected as abnormal, PROVNET checks if this time window can be correlated with any anomalous queue through the overlapping abnormal nodes. Formally, a new abnormal time window $TW_{abnormal}$ can be correlated with, and be appended to an existing queue, $Q$, if it satisfies:

$$\exists TW_i \in Q : AN_{TW_{abnormal}} \cap AN_{TW_i} \neq \emptyset$$

where $AN_{TW}$ is the abnormal nodes in a time window. If $TW_{abnormal}$ cannot be correlated with any existing anomalous queue, PROVNET initializes a new queue for this time window.

While PROVNET is maintaining queues, it also assigns and updates each queue’s anomaly scores to detect malicious time windows. The anomaly score of a queue is computed using the following equations:

$$AS_Q = \prod_{i=1}^{n} (1 + AS_{T_i}) \quad (3.9)$$

$$AS_T = \frac{1}{n} \sum_{\text{Abnormal Edge } e} AS(e) \quad (3.10)$$

In the equations, $AS_Q$ and $AS_T$ are the anomaly scores of a queue and a time window, respectively. $AS(e)$ is the anomaly score of an edge.

The anomaly score of a queue is the product of the anomaly scores of all time windows in the queue. Note that we add 1 to ensure the $AS_Q$ is monotonically increasing. PROVNET computes the anomaly score of a time window by averaging the anomaly scores of abnormal edges.

Ultimately, PROVNET flags a queue as malicious if its anomaly score surpasses a manually predefined threshold, which is set by the security analysts according
to the practical detection requirements (a larger threshold may cause more False Negative while a smaller threshold may cause more false positives). The detection report will document all anomalous time windows, edges, and nodes within the malicious queues.

### 3.8 Intrusion Report Generation

A good intrusion report should not only record the malicious events and system entities with few or no (ideally) false positives but also reduce the human effort to interpret and understand the malicious information flow. An effective way is to visualize the attack behaviors while reducing the size of the attack graph. Given the abnormal edges and nodes in a detection report, PROVNET reconstructs an attack graph and applies a graph reduction technique [57] to reduce the graph size. Specifically, PROVNET summarizes edges that have the same edge types, source, and destination nodes. Figure 3.5 is a simple example of our summarization results.

**Before summarization**

![Before summarization diagram](image)

**After summarization**

![After summarization diagram](image)

**Figure 3.5:** A simple example of the summarization results.
Such visualized attack graphs can help security analysts understand how attackers penetrate the systems and figure out the assets influenced or exfiltrated by the attackers.
Chapter 4

Evaluation

To comprehensively evaluate the effectiveness, efficiency, and robustness of PROVNET, we focus on the following four research questions:

RQ1: How is the detection performance of PROVNET compared to state-of-the-art PIDSes at the time-window level and node level?

RQ2: How do hyper-parameter settings impact the performance of PROVNET?

RQ3: How is the end-to-end run-time performance of PROVNET?

RQ4: Is PROVNET robust to state-of-the-art provenance-based evasion techniques?

4.1 Implementation and Evaluation Settings


To evaluate the performance of PROVNET, we select four publicly available datasets from DARPA’s Transparent Computing Engagement 3 program [28], including several APT attack scenarios (Section 4.2). The evaluations are conducted on a Ubuntu server with 3.1938 GHz AMD EPYC 7343 16-Core Processors, NVIDIA A100 GPU, and 1024 GB memory. PROVNET runs with the default hyper-parameter
settings in the evaluations of detection performance (Section 4.3), run-time performance (Section 4.5), and the robustness against evasion attacks (Section 4.6).

The default settings of hyper-parameters are: the dimension of node feature vector $|\Phi| = 128$, the dimension of node state $|s(\nu)| = 100$ and the size of the sampled neighborhood $|N| = 20$ for TGN learning, the dimension of node embedding $|Z| = 64$, and the size of a time window for detection $|TW| = 15$ minutes.

4.2 Dataset

The evaluated datasets are collected and published by DARPA’s Transparent Computing Engagement 3 program (DARPA TC E3). In the program, DARPA established enterprise network environments and organized red teams and blue teams to simulate APT attack scenarios. The program was hosted for 2 weeks. For preparations, blue teams set up the event logging infrastructure to capture system events for auditing. In addition, the blue team members also simulated various user behaviors that are common in an enterprise as normal data for two weeks. Activities include browsing the web, downloading and installing software, checking email, connecting SSH, etc. Red team members performed attacks on the hosts following the APT attack chain.

According to the attack-targetted hosts, there are five datasets in the engagement: THEIA, CADETS, TRACE, FiveDirections, and CLearScope. THEIA, CADETS, and TRACE record the system events on Linux/FreeBSD, FiveDirections records Windows system events, and ClearScope records the activities on Android platforms. We exclude ClearScope in our evaluations because some dependency semantics within Android platforms are different from Linux/FreeBSD/Windows systems. For example, in the Android platform, a process can write to or read from other processes, while these semantics are out of the scope of our provenance graph construction described in Section 3.4. We leave the detection on Android platforms as future work.

Here, we introduce the attacks performed on the four datasets.

**THEIA.** An attacker exploited a backdoor vulnerability of the Firefox version 54.0.1. After Firefox browsed a malicious website, the backdoor was triggered and the attacker sent a malicious executable script to the victim host. The attacker
then executed the malicious script to obtain a shell with escalated privileges. The attacker used the shell with the root privilege to download another malicious script. Finally, the attacker used it to exfiltrate system information (including user information and system logs) to a remote host and closed the connection to finish the attack.

**CADETS.** An attacker exploited the backdoor of Nginx to download malicious scripts. Then the malicious scripts were executed to elevate the privilege and returned a root-privilege shell to the attacker. The attacker used the shell to download a malicious library from a remote Command and Control (C&C) server. The attacker then attempted to inject into a ssh process but the injection failed and caused the crash of the kernel. The attack ended due to the crash of the system.

**TRACE.** An attacker exploited the backdoor of a Firefox. The backdoor was triggered by browsing a malicious website. The attacker exploited the backdoor to download and execute an in-memory malicious payload, which leaked information in `/etc/passwd` to the attacker. Afterward, the attacker attempted to attack a cache process to elevate the privilege but failed, causing the crash of the system.

**FiveDirections.** Similar to TRACE, an attacker exploited a Firefox backdoor to execute an in-memory malicious payload, which communicates with a remote C&C server. The attacker further used a reconnaissance tool to exfiltrate some sensitive information from the host. However, the attack ended soon because Firefox crashed during the exfiltration and the connection was lost.

<table>
<thead>
<tr>
<th>Dataset</th>
<th># of Nodes</th>
<th># of Edges (in millions)</th>
<th># of Attack Nodes</th>
<th>% of Attack Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CADETS</td>
<td>178,954</td>
<td>10.2</td>
<td>25</td>
<td>0.0140%</td>
</tr>
<tr>
<td>THEIA</td>
<td>443,978</td>
<td>29.0</td>
<td>32</td>
<td>0.0072%</td>
</tr>
<tr>
<td>TRACE</td>
<td>506,424</td>
<td>36.0</td>
<td>511</td>
<td>0.1009%</td>
</tr>
<tr>
<td>FiveDirections</td>
<td>92,569</td>
<td>10.9</td>
<td>385</td>
<td>0.4159%</td>
</tr>
</tbody>
</table>

Table 4.1: Summary of the experimental datasets.

Table 4.1 summarizes the statistics of each dataset. We label the dataset according to the documentation from DARPA. Specifically, we extract the attack-related strings and the start and the stop time of the attacks from the documentation. Based on the attack duration, we figure out the nodes with their attributes including the
key strings. Then we extract 1-hop neighbors of these nodes. According to attack
descriptions in the documentation, we filter nodes irrelevant to attacks and man-
ually reconstruct the attack graphs. All nodes in the attack graphs are labeled as
ground truth for node-level evaluation. All time windows containing attack nodes
are labeled as ground truth for time-window-level evaluation.

Note that while analyzing the THEIA dataset, we found some attack behaviors
that are not documented. The basis for the judgment is that these attack behaviors
include the same malicious scripts, malicious remote host IP, and the same exfil-
tration process as those in the documented attacks. The provenance graph of these
attack behaviors also overlaps with an attack subgraph. What surprises us is that
there is no prior work discovering and discussing this issue in their evaluations.
Since they are indeed attack behaviors, we also label their nodes as node-level
ground truth. The time windows containing attack behaviors are also labeled. We
finish labeling them before the evaluations.

For the evaluations, Table 4.2 shows how we split the dataset for training, val-
idation, and testing. We use only benign data on the dates before the attacks as
training and validation sets.

<table>
<thead>
<tr>
<th>Datasets</th>
<th>Training Data (yyyy-mm-dd)</th>
<th>Validation Data (yyyy-mm-dd)</th>
<th>Test Data (yyyy-mm-dd)</th>
</tr>
</thead>
</table>
| CADETS     | 2018-04-03/04/05          | 2018-04-02                   | 2018-04-06
|            |                            | 2018-04-07                   | 2018-04-11            |
| THEIA      | 2018-04-03/04/05          | 2018-04-09                   | 2018-04-10            |
|            |                            | 2018-04-11                   |                       |
| TRACE      | 2018-04-03/04/05          | 2018-04-07                   | 2018-04-10            |
|            |                            | 2018-04-11                   |                       |
| FiveDirections | 2018-04-03/04/05/06/07/08 | 2018-04-09                   | 2018-04-10
|            |                            | 2018-04-11                   |                       |

Table 4.2: DARPA data used for training, validation, and test. The bold days
are attack days in which both benign and attack activities exist. The remaining
days are benign days with only benign activities.
4.3 Detection Performance

To better measure the detection performance of PROVNET, we aim to compare PROVNET with state-of-the-art PIDSes. However, fair comparisons are not easy when benchmarking the PIDSes. PROVNET is an anomaly-based PIDS that performs detection at time-window and node levels. We do not consider any signature-based detectors [35, 39, 40] because their performance relies on unpublished signatures from expert knowledge. Signatures written by ourselves can bring biases during the evaluations.

For the anomaly-based detectors, we select the following state-of-the-art PIDSes for our evaluations (because they are either publicly available or easy to re-implement): SIGL [20], ThreaTrace [54] and Kairos [6]. Note that SIGL was originally designed to detect anomalies in software installation graphs, which are much smaller than system provenance graphs. Its graph LSTM encoder incurs high computational complexity when processing a large graph and cannot learn the representation of large graphs well [15]. So we follow prior work [15] to re-implement a SIGL-like PIDS, which replaces the graph LSTM with GNNs that can learn from large graphs and everything else keeps the same. For ThreaTrace, as it performs detection at the node level, we compare it only in node-level evaluations. Kairos can work on both time-window and node-level detection, so we evaluate it in both.

4.3.1 Time-window-level Evaluation

For time-window-level evaluation, we collect the following metrics:

\[
\text{Precision} = \frac{TP}{TP + FP}
\]

\[
Recall = \frac{TP}{TP + FN}
\]

\[
F1\text{-score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}
\]

\[
\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}
\]

The TP, TN, FP, and FN stand for the numbers of true positives, true negatives, false positives, and false negatives counted at the time-window level respectively.
Table 4.3: Time-window-level Experimental Results. The bold values show the best results in the datasets.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Precision</th>
<th>Recall</th>
<th>F1-Score</th>
<th>Accuracy</th>
<th>TN</th>
<th>FP</th>
<th>FN</th>
<th>TP</th>
</tr>
</thead>
<tbody>
<tr>
<td>CADETS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PROVNET</td>
<td><strong>1.00000</strong></td>
<td>0.75000</td>
<td>0.85714</td>
<td>0.99441</td>
<td>175</td>
<td>0</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Kairos</td>
<td><strong>1.00000</strong></td>
<td>1.00000</td>
<td>1.00000</td>
<td><strong>1.00000</strong></td>
<td>175</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>SIGL</td>
<td>0.28571</td>
<td>0.50000</td>
<td>0.36364</td>
<td>0.96089</td>
<td>170</td>
<td>5</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>THEIA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PROVNET</td>
<td><strong>1.00000</strong></td>
<td>0.80000</td>
<td><strong>0.88889</strong></td>
<td><strong>0.99254</strong></td>
<td>129</td>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Kairos</td>
<td>0.05495</td>
<td>1.00000</td>
<td>0.10417</td>
<td>0.35821</td>
<td>43</td>
<td>86</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>SIGL</td>
<td>0.36364</td>
<td>0.80000</td>
<td>0.50000</td>
<td>0.94030</td>
<td>122</td>
<td>7</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>TRACE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PROVNET</td>
<td><strong>1.00000</strong></td>
<td>0.75000</td>
<td>0.85714</td>
<td><strong>0.99425</strong></td>
<td>170</td>
<td>0</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Kairos</td>
<td>N/A</td>
<td>0.00000</td>
<td>N/A</td>
<td>0.97701</td>
<td>170</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>SIGL</td>
<td>0.50000</td>
<td>0.25000</td>
<td>0.33333</td>
<td>0.97701</td>
<td>169</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>FiveDirections</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PROVNET</td>
<td><strong>1.00000</strong></td>
<td>0.66667</td>
<td><strong>0.80000</strong></td>
<td><strong>0.98901</strong></td>
<td>176</td>
<td>0</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Kairos</td>
<td>0.11538</td>
<td>1.00000</td>
<td>0.20690</td>
<td>0.74725</td>
<td>130</td>
<td>46</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>SIGL</td>
<td>0.66667</td>
<td>0.66667</td>
<td>0.66667</td>
<td>0.97802</td>
<td>174</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 4.3 shows the time-window level detection results. For the CADETS dataset, Kairos achieves the best performance. After checking its alerts, we found that Kairos flags an abnormal time window, however through some false positive alerts rather than true positives. Its results on the THEIA and FiveDirections datasets reflect this trend. There are no false negatives but many false positives (86 in the THEIA and 46 in the FiveDirections). However, Kairos obtains the opposite results from the TRACE dataset. There is nothing detected. We further investigate the intermediate artifacts of Kairos during detection. Kairos sets the threshold too high and cannot detect anything. Unlike other datasets, the size of the TRACE dataset is the largest and includes various benign activities. Kairos’ link prediction approach lacks generalizability to those benign activities and hence computes a high threshold for detection.

While SIGL has lower false positives, it cannot well detect abnormal time windows. By comparison, PROVNET achieves good results with no false positives in all datasets. We also notice that PROVNET always has 1 or 2 false negative time windows in each dataset. After manually investigating the false negative time windows, we found that these time windows are at the early stages in the APT attack chain. As introduced early in this chapter, most of the attacks in the dataset start by exploiting the backdoor vulnerability of Firefox or Nginx. In the early exploitation stages, Firefox or Nginx communicates with a remote malicious IP. Such mali-
cious network communication behaviors do not differ much from normal network communication behaviors. So PROVNET cannot identify the difference between benign and attack graphs in these stages. More details about this limitation will be discussed in the discussion chapter (see Chapter 5)

4.3.2 Node-level Evaluation

In node-level evaluations, apart from the metrics measured in time-window-level evaluations (Precision, Recall, F1-score, and Accuracy), we also measure the False Discovery Rate ($FDR = \frac{FP}{TP+FP}$) in particular, where all TP, TN, FP, FN are counted at the node level. This metric measures the ratio of FP nodes to all positive nodes. At a high level, this metric reflects how much extra work is required by security analysts on attack investigations. A smaller value means less human effort. This is a good measure of our design goal: reducing human efforts when interpreting the detection reports and investigating the attacks.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Precision</th>
<th>Recall</th>
<th>F1-Score</th>
<th>Accuracy</th>
<th>FDR</th>
<th>TN</th>
<th>FP</th>
<th>FN</th>
<th>TP</th>
</tr>
</thead>
<tbody>
<tr>
<td>CADETS</td>
<td>PROVNET</td>
<td>1.00000</td>
<td>0.72614</td>
<td>0.90298</td>
<td>0.99982</td>
<td>0.00000</td>
<td>113745</td>
<td>0</td>
<td>21</td>
</tr>
<tr>
<td>Kairos</td>
<td>0.07556</td>
<td>0.56000</td>
<td>0.23962</td>
<td>0.84976</td>
<td>0.923414</td>
<td>113477</td>
<td>268</td>
<td>3</td>
<td>22</td>
</tr>
<tr>
<td>ThreaTrace</td>
<td>0.00182</td>
<td>0.00800</td>
<td>0.00356</td>
<td>0.999016</td>
<td>0.99818</td>
<td>112649</td>
<td>1096</td>
<td>23</td>
<td>2</td>
</tr>
<tr>
<td>SIGL</td>
<td>0.00086</td>
<td>0.00800</td>
<td>0.00169</td>
<td>0.99727</td>
<td>0.99914</td>
<td>111409</td>
<td>2336</td>
<td>23</td>
<td>2</td>
</tr>
<tr>
<td>THEIA</td>
<td>PROVNET</td>
<td>1.00000</td>
<td>0.87719</td>
<td>0.99997</td>
<td>0.00000</td>
<td>253337</td>
<td>0</td>
<td>7</td>
<td>25</td>
</tr>
<tr>
<td>Kairos</td>
<td>0.00022</td>
<td>0.96875</td>
<td>0.00044</td>
<td>0.44479</td>
<td>0.99978</td>
<td>112065</td>
<td>140872</td>
<td>1</td>
<td>31</td>
</tr>
<tr>
<td>ThreaTrace</td>
<td>0.00096</td>
<td>0.21875</td>
<td>0.00192</td>
<td>0.97124</td>
<td>0.99904</td>
<td>246115</td>
<td>725</td>
<td>25</td>
<td>7</td>
</tr>
<tr>
<td>SIGL</td>
<td>0.26667</td>
<td>0.12500</td>
<td>0.17062</td>
<td>0.99985</td>
<td>0.73333</td>
<td>253326</td>
<td>11</td>
<td>28</td>
<td>4</td>
</tr>
<tr>
<td>TRACE</td>
<td>PROVNET</td>
<td>1.00000</td>
<td>0.78735</td>
<td>0.99771</td>
<td>0.00000</td>
<td>201033</td>
<td>0</td>
<td>46</td>
<td>50</td>
</tr>
<tr>
<td>Kairos</td>
<td>0.00000</td>
<td>0.99946</td>
<td>N/A</td>
<td>N/A</td>
<td>0.99771</td>
<td>94556</td>
<td>106477</td>
<td>267</td>
<td>244</td>
</tr>
<tr>
<td>ThreaTrace</td>
<td>0.00259</td>
<td>0.47750</td>
<td>0.00455</td>
<td>0.47037</td>
<td>0.99771</td>
<td>201033</td>
<td>0</td>
<td>511</td>
<td>0</td>
</tr>
<tr>
<td>SIGL</td>
<td>0.88889</td>
<td>0.01566</td>
<td>0.03077</td>
<td>0.99750</td>
<td>0.11111</td>
<td>201032</td>
<td>1</td>
<td>503</td>
<td>8</td>
</tr>
<tr>
<td>FiveDirections</td>
<td>PROVNET</td>
<td>1.00000</td>
<td>0.73996</td>
<td>0.90932</td>
<td>0.00000</td>
<td>304018</td>
<td>0</td>
<td>333</td>
<td>52</td>
</tr>
<tr>
<td>Kairos</td>
<td>0.01871</td>
<td>0.49610</td>
<td>0.03066</td>
<td>0.70317</td>
<td>0.98129</td>
<td>24000</td>
<td>10018</td>
<td>194</td>
<td>191</td>
</tr>
<tr>
<td>ThreaTrace</td>
<td>0.05512</td>
<td>0.01818</td>
<td>0.02734</td>
<td>0.98553</td>
<td>0.94488</td>
<td>33908</td>
<td>120</td>
<td>378</td>
<td>7</td>
</tr>
<tr>
<td>SIGL</td>
<td>0.25000</td>
<td>0.00260</td>
<td>0.00514</td>
<td>0.98875</td>
<td>0.75000</td>
<td>30415</td>
<td>3</td>
<td>384</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.4: Node-Level Experimental Results. The bold values show the best results in the datasets.

Table 4.4 shows the evaluation results of systems at the node level. Similar to time-window results, Kairos flags many false positives in CADETS, THEIA, and FIveDirections datasets. For the TRACE dataset, as explained above, it cannot detect anything due to the high threshold. For ThreaTrace, as it reports abnormal nodes and their 2-hop neighbors, it also flags many false positives. Similar
to time-window results, SIGL has the worst detection performance. By comparison, PROVNET achieves good detection performance. Remember that our goal is to detect malicious behaviors while reducing false positives. From the FDR column, PROVNET has 0s results in all datasets, indicating that security analysts can dedicate themselves to understanding the attacks without the bother of false positive alerts. In contrast, other state-of-the-art systems have high FDR results, especially Kairos and ThreaTrace. Most of their FDRs are larger than 90% and some even reach 99%. This means when analyzing their detection reports, at least 90% of the time will be wasted on useless false positive alerts, which is unacceptable in practice.

However, since PROVNET cannot detect some abnormal time windows, all malicious nodes within these time windows cannot be detected. PROVNET hence has low Recall results in some datasets. Even though, given the abnormal nodes detected in further stages, security analysts can leverage some backtracking techniques \([11, 30]\) to recover the whole attack graphs during attack investigation.

### 4.4 Hyper-parameter Impacts on Performance

We select the THEIA dataset for our hyper-parameter and robustness evaluations and limitation discussions because this dataset contains the complete APT attack chain. In addition, the attacks in the THEIA do not crash the systems, meaning that all recorded logs are complete. The attacks in other datasets either cause the crashes of the systems or the services, so their attack chains lack the last several stages.

We measure the impact of a hyper-parameter on performance by varying its value while other hyper-parameters use the default settings. Figure 4.1 and Figure 4.2 present the impacts on time-window-level and node-level detection performance, respectively. Figure 4.3 presents the impacts on memory usage at run-time.
Figure 4.1: Hyper-parameter impacts on time-window-level detection performance. The y-axis is the evaluated metrics. The x-axis is the values of hyper-parameter settings.
Figure 4.2: Hyper-parameter impacts on node-level detection performance. The y-axis is the evaluated metrics. The x-axis is the values of hyper-parameter settings.
Figure 4.3: Hyper-parameter impacts on runtime performance. The y-axis is the average memory usage in megabytes, and the x-axis is the values of hyper-parameter settings.
Node Feature Vector Dimension $|\Phi|$. $|\Phi|$ encodes how much contextual information about a node is learned by the word2vec model. A higher dimension allows for a more nuanced representation of node characteristics. From the first subfigure in Figure 4.1 and Figure 4.2, as $|\Phi|$ increases, there is initially an improvement in performance metrics due to the more detailed feature representations, which provide more information for the model to learn the node features. When $|\Phi|$ reaches 128, there are no changes in the detection results, meaning that this dimension is enough to encode all information of a node. However, memory usage increases with larger $|\Phi|$, because a higher-dimensional feature requires more memory space.

Node State Dimension $|s(v)|$. $|s(v)|$ stores the historical information of how a node’s neighbors change over time. Increasing $|s(v)|$ improves the detection performance initially by allowing more history information to be encoded. However, performance can degrade beyond a certain dimension size ($|s(v)| > 100$), because some redundant history information is also encoded, which causes the noise to negatively influence the model performance. A larger $|s(v)|$ increases the memory usage as the model must allocate more space to store and process the expanded state information for each node.

Neighborhood Size $|N|$. $|N|$ specifies the size of the neighborhood sampled for GNN learning. Two nodes are more likely to have the same roles if they have similar neighborhood structures. So as $|N|$ increases, PROVNET can leverage more neighborhood information to judge the roles of nodes within a provenance graph. Shown in Figure 4.1 and Figure 4.2, when $|N|$ is larger than 20, there is no improvement in the detection performance. That is because the majority of nodes in the THEIA dataset have neighborhoods with fewer than 20 nodes. Increasing $|N|$ does not bring more neighborhood information to learn. Memory usage reflects this fact. From the third subfigure in Figure 4.3, when $|N|$ is larger than 20, there is little increase in memory consumption, meaning that only a few nodes have neighborhoods larger than 20.

Node Embedding Dimension $|Z|$. $|Z|$ determines the size of the vector space in which temporal and structural information is embedded, impacting the model’s capacity to represent and differentiate nodes. From the figures, although we set 64 as the default setting, we can see that $|Z| = 16$ can encode sufficient information to obtain good detection performance. Meanwhile, $|Z| = 16$ also consumes less
memory, which is the best choice for detection. When \( |Z| \) is larger than 64, the detection performance decreases, because a large \( |Z| \) decreases the generalizability of the model. When \( |Z| \) is set to be too large, the model may not generalize well and could overfit the training data, failing to capture the underlying structure or distribution of the data.

**Time Window Size \(|TW|\).** \( |TW| \) shows how frequently PROVNET performs detection. From the figures, although the time-window-level performance is changed as \( |TW| \) increases, the node-level performance remains the same. This means PROVNET may miss some abnormal nodes in some time windows, but still can detect them from other time windows. For example, an extreme case is that there is only an abnormal event at the end of a time window and other malicious operations are in the next time window. Such extreme imbalance of data distribution causes PROVNET unable to identify the abnormal nodes as there is not enough information for judgment. However, if \( |TW| \) is set to be too large, it will consume more computing resources and may require more time to finish the detection on each time window.

In summary, each hyper-parameter affects both the detection performance and memory usage of PROVNET. While larger values for some parameters can improve PROVNET’s ability to detect anomalies, they also increase the computational load. So security analysts can tune the hyper-parameters to obtain the optimal performance based on their computing resources.

### 4.5 End-to-end Run-time Performance

One important requirement for designing PIDS is the end-to-end run-time performance. Intrusion detection systems are often required to operate in real-time to promptly detect and respond to malicious activities. If the run-time performance is sluggish, the system may fail to detect intrusions quickly enough, leading to potential security breaches. Thus, we evaluate the end-to-end run-time performance of PROVNET on the THEIA dataset and plot the results in Figure 4.4.

In general, PROVNET takes fewer than 70 seconds to finish end-to-end detection on a 15-minute time window. In the worst case, PROVNET takes at most 273 seconds to detect a 15-minute time window with 2,529,280 edges, which is well
Figure 4.4: End-to-end run-time performance of PROVNET on the THEIA dataset. The y-axis is the time consumed to detect a time window. The x-axis is the tested time window below the size of the time window. The median size of the time window contains 61,440 edges, and PROVNET takes only 11.3 seconds to finish detection. Given such run-time performance, PROVNET is able to detect anomalies in real-time, allowing security analysts to execute appropriate countermeasures swiftly to mitigate any damage caused by an intrusion.

4.6 Robustness

We next explore the robustness of PROVNET against evasion attacks. The experiments are conducted to test PROVNET’s ability to maintain high detection accuracy in the face of sophisticated attacks that are well-orchestrated to bypass detection. We use two state-of-the-art provenance-based evasion attacks [15, 41] for the robustness evaluations. At a high level, both approaches try to simulate some specific operations and change the attack graph structure to evade detection. To do so, they sample some benign events, insert them into the attack graphs according to their evasion algorithms, and generate the evasion graphs. As these approaches follow the information flow semantics of provenance graphs, the evasion graphs can be generated in practice through some well-designed attack behaviors. The THEIA dataset is used for evaluation as one of the evasion techniques is designed for this dataset.
4.6.1 Mimicry Attack

Goyal et al [15] propose a Mimicry Attack to evade state-of-the-art PIDSes. Their approach is based on the fact that the neighborhood structure of the attack nodes influences the node embeddings generated by PIDSes’ models.

Thus, this Mimicry Attack aims to reduce the difference between node embeddings of attack and benign nodes. To do so, Mimicry Attack firstly samples benign subgraphs. Then it repeatedly inserts benign subgraphs into the neighborhood of attack nodes. As such, the neighborhoods of attack nodes are similar to the neighborhoods of benign nodes. During the generation of the node embeddings, the differences between attack and benign node embeddings become smaller, thereby bypassing the detection.

Figure 4.5 shows how Mimicry Attack is performed on THEIA dataset. In the THEIA dataset, Firefox is exploited by the attackers. The attackers first sample some benign Firefox subgraphs. Then these subgraphs are repeatedly injected into the neighborhood through clone operations. Such clone operations semantically ensure the correctness of the evasion graph, so this evasion graph can be collected in practice.

Evaluation Protocols. We use the codes open-sourced by the authors to generate the evasion graphs. We collect the benign Firefox subgraphs from the training set. Since PROVNET processes provenance graphs with temporal information, we forge timestamps for all inserted benign Firefox subgraphs. Considering that the attacker repeatedly triggers the Firefox benign behaviors during the attack to hide the malicious activities, all forged timestamps are within the attack period. During evasion graph generations, we set up a hyper-parameter to control the number of the inserted benign Firefox subgraphs. We generate multiple evasion sets with different numbers of benign Firefox subgraphs inserted for our evaluations.

Here are some statistics about the generated evasion sets:

- The number of edges in Firefox benign subgraphs: 5,821,142.
- The number of edges in the original testing set: 13,643,067.
- The number of edges in the evasion graph (1 benign subgraph inserted): 19,464,209 (13,643,067 + 1 * 5,821,142).
Figure 4.5: The *Mimicry Attack* on the THEIA dataset. In the graph, circles are process nodes, rectangles are file nodes, and diamonds are network socket nodes. Red nodes are attack-related, green nodes are benign, and orange nodes are benign nodes inserted by the *Mimicry Attack*. 
• The number of edges in the evasion graph (3 benign subgraphs inserted): 31,106,493 (13,643,067 + 3 * 5,821,142).

• The number of edges in the evasion graph (5 benign subgraphs inserted): 42,748,777 (13,643,067 + 5 * 5,821,142).

• The number of edges in the evasion graph (7 benign subgraphs inserted): 54,391,061 (13,643,067 + 7 * 5,821,142).

• The number of edges in the evasion graph (9 benign subgraphs inserted): 66,033,345 (13,643,067 + 9 * 5,821,142).

Note that when the number of benign subgraphs inserted is greater than or equal to 3, the number of added edges is greater than the original testing set, which means the attack subgraphs have been hidden within a large number of benign subgraphs.

**Evaluation results.** Figure 4.6 shows the evaluation results on all evasion graphs. From the figure, we can see that PROVNET’s detection performance does not change regardless of the number of benign edges added.

The *Mimicry Attack* can evade some graph-level (e.g., StreamSpot [37], Unicorn [19]) or path-level (e.g., ProvDetector [53], Pagoda [56]) PIDSes because they generate the graph or path embeddings and perform detection at graph or path granularities. As the number of benign subgraphs increases, the attack subgraphs (or paths) account for a smaller proportion of the graph (or paths) and contribute less to graph (or path) embedding generation. So the difference between benign and attack embeddings reduces accordingly.

However, PROVNET performs detection at the node level. Unlike those graph or path embedding approaches, when increasing the number of added benign subgraphs, the attack nodes still contribute a lot when generating their node embeddings, retaining the distinction between benign node embeddings. In addition, PROVNET generates the node embeddings by learning the historical changes of a node’s neighborhood. From Figure 4.5, we can see that the malicious Firefox may be influenced by those inserted benign nodes when generating the embeddings. By comparison, the malicious processes (profile and clean) are not influenced too much because they mainly interact with the system files and the malicious remote
Figure 4.6: The evaluation results on the evasion graphs generated through Mimicry Attack. The y-axis is the evaluated metrics. The upper x-axis is the number of benign subgraphs inserted; the lower x-axis is the number of benign edges inserted (in millions).

IP. So their node embeddings are still dominated by malicious interactions when generated. Thus, PROVNET is robust to this Mimicry Attack.

4.6.2 ProvNinja

Mukherjee et al propose ProvNinja [41] to evade the provenance-based machine learning detector (e.g., ProvDetector [53], ShadeWatcher [58]). Different from Mimicry Attack that modifies the neighbors of attack nodes, ProvNinja modifies the attack subgraph itself. It aims to replace some malicious edges with some common benign provenance paths. As such, the attack subgraph is similar to the benign subgraphs and can evade detection.

To achieve this, ProvNinja initially computes the frequency scores for historical benign edges and establishes a frequency database. Subsequently, it identifies
rare events in attack graphs with their source and destination nodes. *ProvNinja* proceeds to verify if the source and destination nodes of the rare malicious edges are present in the frequency database. Should these nodes exist in the database, *ProvNinja* then generates common paths (evasion gadgets) from the source to the destination nodes, utilizing the frequency scores of benign edges. Ultimately, these rare malicious edges are substituted with the evasion gadgets.

Figure 4.7: *ProvNinja* on the THEIA dataset. The upper graph is the original attack subgraph. The graph at the bottom is the simplified evasion graph generated by *ProvNinja*. In the graphs, circles are process nodes, rectangles are file nodes, and diamonds are network socket nodes. Red nodes are attack-related, green nodes are benign, and orange nodes are benign nodes inserted by the *ProvNinja*.

Figure 4.7 shows the evasion subgraph generated by *ProvNinja*. In the figure,
the only source and destination nodes that are in the frequency database are Firefox
and /dev/glx.alsa_675. So even though the edges between them are not as rare as
some other malicious edges, they are still replaced with the orange benign paths.
The other part retains that same structure.

**Evaluation Protocols.** We use the code open-sourced by the authors to gener-
ate the evasion graphs. We construct the frequency databases with the edges in
the training set. Note that ProvNinja removes all temporal information from the
provenance graphs. So we make the adjustment that we assign time information of
evasion gadgets using the time information of the replaced edges. Formally, given
an attack edge $e_{\text{attack}}$ and its evasion gadgets $g$:

$$\forall e \in g : Time(e) = Time(e_{\text{attack}})$$  \hspace{1cm} (4.1)

**Evaluation results.** Table 4.5 presents the detection performance of PROVNET on
the original testing set and the evasion set. PROVNET has the same performance
on both sets, indicating that ProvNinja cannot evade the detection from PROVNET.

<table>
<thead>
<tr>
<th>Testing set</th>
<th>Precision</th>
<th>Recall</th>
<th>F1</th>
<th>Accuracy</th>
<th>TN</th>
<th>FP</th>
<th>FN</th>
<th>TP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>1.0</td>
<td>0.78125</td>
<td>0.87719</td>
<td>0.99997</td>
<td>253,337</td>
<td>0</td>
<td>7</td>
<td>25</td>
</tr>
<tr>
<td>ProvNinja</td>
<td>1.0</td>
<td>0.78125</td>
<td>0.87719</td>
<td>0.999997</td>
<td>2,090,439</td>
<td>0</td>
<td>7</td>
<td>25</td>
</tr>
</tbody>
</table>

**Table 4.5:** Comparison of detection performance between the original and
ProvNinja testing sets.

There are two reasons why PROVNET is robust to ProvNinja:

First, the core of ProvNinja focuses on the frequency of events. It tries to
generate the evasion graphs by replacing some rare malicious events with some
common events (which frequently exist in the past). This approach can evade some
PIDSes relying on frequency-based approaches, e.g. ProvDetector [53]. PROVNET,
however, does not use any frequency information for detection. So the generated
evasion gadgets do not influence much on PROVNET ’s detection performance.

Second, ProvNinja can generate the evasion gadgets only for the source and
destination nodes that have been seen in the past. In APT attacks, malicious files or
processes might not be shown in the benign historical data (profile and clean shown
in Figure 4.7). So, the malicious events between these processes are not replaced.
Therefore, from Figure 4.7, we can see that there is still a large proportion of the attack subgraphs kept in the evasion graph. Those attack subgraphs are sufficient enough for PROVNET’s detection.
Chapter 5

Discussion

5.1 Limitations

Limitation 1. From the evaluation of detection performance (Section 4.3), we see that PROVNET is unable to detect any attack in the early stages of APT attacks. To understand the reason, we manually investigate the neighborhoods of the attack nodes.

Figure 5.1 shows that the benign neighbors are highly similar to the attack subgraph missed by PROVNET in structure. The attack subgraph missed by PROVNET maps to the Initial Reconnaissance and Initial Compromise stages of the APT attacks. During these stages, malicious payloads are sent to Firefox to exploit the backdoor and download the malicious scripts. From the perspective of Firefox, it communicates with a remote IP (malicious), loads some libraries for execution, and writes files (malicious executables) to the disk. Structure-wise, this is identical to the benign behaviors of Firefox, which communicates with a remote IP (benign), loads some libraries for execution, and writes files (temporary files) to the disk. Without further execution of malicious executables, PROVNET cannot identify the difference between the attack and benign subgraphs, because the historical changes of the neighborhood are almost the same. PROVNET lacks enough history information to judge if the subgraph is abnormal or not. So, as PROVNET continues to learn the evolving patterns of malicious nodes, it can only detect the following abnormal subgraph structure.
Figure 5.1: The attack subgraphs detected and missed by PROVNET. The red nodes are attack-related. The green nodes are benign.

There is also a concern that PROVNET performs well because the distributions in the graphs cause the overfitting issue. Randomness in random walk ensures that PROVNET does not overfit the training set. However, if most of the nodes only have one out-degree in the datasets, the randomness degree decreases, which causes PROVNET to overfit the training data.

To address this concern, we measure the out-degree distributions of nodes in datasets. Here are the results:

- **CADETS.** 0.9% of nodes do not have any out-degrees; 42.9% of nodes have only one out-degree; 52.7% of nodes have 2 to 10 out-degrees; 1.7% of nodes have 11 to 20 out-degrees; 1.8% of nodes have more than 20 out-degrees.

- **THEIA.** 21.2% of nodes do not have any out-degrees; 17% of nodes have only one out-degree; 54.9% of nodes have 2 to 10 out-degrees; 1.4% of nodes have 11 to 20 out-degrees; 5.6% of nodes have more than 20 out-degrees.
• **TRACE.** 14.4% of nodes do not have any out-degrees; 36.6% of nodes have only one out-degrees; 32.6% of nodes have 2 to 10 out-degrees; 11.8% of nodes have 11 to 20 out-degrees; 4.6% of nodes have more than 20 out-degrees.

• **FiveDirections.** 15.9% of nodes do not have any out-degrees; 31% of nodes have only one out-degrees; 38.2% of nodes have 2 to 10 out-degrees; 2.6% of nodes have 11 to 20 out-degrees; 12.3% of nodes have more than 20 out-degrees.

Results show that, in all datasets, more than half of the nodes have at least two out-degrees or do not have any descendants. They still ensure randomness during model training and prevent PROVNET from overfitting the training data.

**Limitation 2.** Another limitation of PROVNET is that it seems unable to learn the representation of a large graph well. Apart from DARPA TC E3 datasets, we also evaluated PROVNET with DARPA TC E5 datasets. Table 5.1 shows that evaluation results of PROVNET on DARPA E5 datasets.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Precision</th>
<th>Recall</th>
<th>F1-Score</th>
<th>Accuracy</th>
<th>TN</th>
<th>FP</th>
<th>FN</th>
<th>TP</th>
</tr>
</thead>
<tbody>
<tr>
<td>CADETS</td>
<td>1.00000</td>
<td>0.90698</td>
<td>0.95122</td>
<td>0.99989</td>
<td>148395</td>
<td>0</td>
<td>16</td>
<td>156</td>
</tr>
<tr>
<td>THEIA</td>
<td>N/A</td>
<td>0</td>
<td>N/A</td>
<td>0.99992</td>
<td>405232</td>
<td>0</td>
<td>34</td>
<td>0</td>
</tr>
<tr>
<td>TRACE</td>
<td>N/A</td>
<td>0</td>
<td>N/A</td>
<td>0.99998</td>
<td>789199</td>
<td>0</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>FiveDirections</td>
<td>N/A</td>
<td>0</td>
<td>N/A</td>
<td>0.99947</td>
<td>287384</td>
<td>0</td>
<td>151</td>
<td>0</td>
</tr>
</tbody>
</table>

**Table 5.1:** Node-Level detection results of PROVNET on DARPA E5 datasets.

We notice that PROVNET cannot detect anything except in the CADETS dataset. Since PROVNET is not designed with explainability, it is hard to figure out the reasons for these results. One potential reason may be due to the graph size. In E5 data, only CADETS datasets have the same size as E3 datasets, while others are much larger than E3 datasets. The detection performance of CADETS E5 seems to verify this assumption, but we cannot establish a solid analysis for it currently. Note that the attacks in the THEIA E5 dataset crash the system in the early stage of the APT chain. So, PROVNET cannot detect anything in this dataset either.
5.2 Future Work

**Improve Detection Performance.** As described in the limitation section, PROVNET is unable to detect the malicious behaviors in the early stages of APT attacks. A possible research direction can be leveraging the back-tracking techniques [11, 30] to attribute the entries of the attacks. However, this inevitably incurs extra overhead for intrusion detection. Thus, there is also a trade-off between detection and run-time performance.

In addition, PROVNET is not designed for the Android platforms. So another improvement is to incorporate Android information flow analysis [14, 50] and provenance-based intrusion detection.

**Explainability.** PROVNET’s detection performance relies on the performance of the graph autoencoder. Current designs do not consider explainability, hence hindering our understanding of how PROVNET makes a decision on anomalies. Recent work [18] attempts to interpret the decision-making of an edge-based detector. It uses an optimization-based approach to pinpoint benign reference edges that are nearest in the feature space to the anomalous detection signals. The understanding of anomalous detection signals is converted into understanding the features of reference edges. PROVNET may be beneficial with such a reference-based approach to explaining the decision-making process of a node, thereby trying to understand why PROVNET cannot detect any anomalies in DARPA E5 data.

**Robustness.** We did not actively consider the robustness of PROVNET when designing it. It is robust against state-of-the-art provenance-based evasion attacks because the designs of these evasion approaches are trivial. They only target evading a small subset of detection mechanisms (e.g., graph-/path-level detectors or frequency-based detectors). However, in our threat model, we assume the attackers have the knowledge of the design details of PROVNET, which means they can also devise evasion techniques specific to evade PROVNET. Therefore, an effective way to actively improve the robustness of PROVNET is a research direction worth exploring.
Chapter 6

Conclusion

We systematically conduct evaluations to investigate human efforts to interpret detection reports for attack attributions. The results show that some state-of-the-art PIDSes only alert the abnormality of a graph without providing more fine-grain information in the detection reports; the others alert specific abnormal system entities to security analysts but tend to include redundant information or generate many false positive alerts, causing fatigue or analyst burnout. To address the limitations, we present the PROVNET. It uses a semantics-enriched NLP technique to featurize provenance graphs and leverages a graph autoencoder with temporal graph learning framework to learn the graph representation and alert anomalies. PROVNET further removes the false positives by removing the temporary files and correlating the abnormal time windows. Finally, PROVNET generates an insightful detection report containing malicious time windows, nodes, edges, and a reconstructed attack chain. We leverage DARPA datasets to evaluate the detection performance, runtime performance, and robustness of PROVNET. Results show that the PROVNET achieves competitive performance compared with state-of-the-art systems. We finish by discussing the limitations of PROVNET and the potential research directions for future work.
Bibliography


59


61


detection: A review. ACM Comput. Surv., 54(2), mar 2021. ISSN
0360-0300. → page 8

[45] T. Pasquier, X. Han, M. Goldstein, T. Moyer, D. Eyers, M. Seltzer, and
J. Bacon. Practical whole-system provenance capture. In Symposium on
Cloud Computing (SoCC’17). ACM, ACM, 2017. → page 6

Large Corpora. In Proceedings of the LREC 2010 Workshop on New

page 1

Temporal graph networks for deep learning on dynamic graphs. arXiv

prediction: Unified message passing model for semi-supervised
classification. In Proceedings of the Thirtieth International Joint Conference
on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada,

Flowmine: Android app analysis via data flow. In 2016 13th IEEE Annual
Consumer Communications & Networking Conference (CCNC), pages
435–441. IEEE, 2016. → page 56

and S. R. Rajagopalan. A human capital model for mitigating security
analyst burnout. In Eleventh Symposium On Usable Privacy and Security

Graph attention networks. In International Conference on Learning
Representations (ICLR’18). OpenReview.net, 2018. → page 7

[53] 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


