IOT INTEGRATION, ADVERSARIAL ATTACKS, AND THREAT EXPLANATIONS IN PROVENANCE-BASED INTRUSION DETECTION SYSTEMS

by

Kunal Mukherjee

APPROVED BY SUPERVISORY COMMITTEE:

Kangkook Jee, Chair

Murat Kantarcioğlu, Co-Chair

Bhavani Thuraisingham

Feng Chen

Copyright © 2024

Kunal Mukherjee

All rights reserved

To

the best parents in the world, Mr. Kingsuk Mukherjee and Dr. Sharmistha Mukherjee,

my eternal headache, my brother, Mr. Spandan Mukherjee,

loving grandparents, Late Dr. Sisir Kumar Mukherjee and Mrs. Anju Mukherjee,

and Dr. Sati Prasad Acharya and Mrs. Subra Acharya.

IOT INTEGRATION, ADVERSARIAL ATTACKS, AND THREAT EXPLANATIONS IN PROVENANCE-BASED INTRUSION DETECTION SYSTEMS

by

KUNAL MUKHERJEE, BS

DISSERTATION
Presented to the Faculty of
The University of Texas at Dallas
in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY IN
COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT DALLAS
May 2025

ACKNOWLEDGMENTS

This dissertation would not have been possible without the unwavering support of my family and friends. "Family" in this context encompasses not just my blood relations but also the incredible people who entered my life and found a place in my heart.

First and foremost, I extend my deepest gratitude to my advisors, Dr. Kangkook Jee and Dr. Murat Kantarcioğlu. Their mentorship, guidance, and endless reservoir of knowledge has been instrumental in shaping both my research and personal growth. I owe them a profound debt of thanks. I would also like to express my appreciation to Dr. Bhavani Thuraisingham and Dr. Feng Chen for serving on my committee and providing invaluable feedback that helped refine my dissertation.

To my friend and colleague, Josh Wiedemeier—thank you for your emotional support, critical feedback, brainstorming sessions, and thoughtful proofreading. Your camaraderie has been a cornerstone in my PhD journey, and I am grateful to have had you as a close confidant throughout this process. I would also be remiss if I didn't acknowledge the countless discussions and collaborations with Tianhao Wang, James Wei, Guangze Zu, Jonathan Wu, Nicholas Baker, Kaerah Lopez, Jerry Teng, Elliot Tarbet, David Wank, Ahad Jawaid, Jaehyun Park, Dr. Sangsoo Ko, Simon Klancher, Zarish Mahboob, and other amazing members of our research group. I consider myself incredibly fortunate to have worked alongside such brilliant minds.

I would like to extend my sincere thanks to my teammates at Zillow Group, Inc. for an enriching internship experience. I want to offer a special mention to Saeid Balaneshin (my manager), Zachary Harrison (my mentor), Dr. Ondrej Linda, Kelsey Juraschka, Jack Gibbons, Dr. Kai Liu, Giuliano Janson, and Dr. Faraz Moghimi. Their guidance and support made my time at Zillow both productive and unforgettable. I also wish to thank my first industrial mentors from Ciholas, Inc., including Mike Ciholas, Justin Bennett (my manager),

Jawad AlDamini (my mentor), Tim DeBaillie, Daniela Fuentes, and Mason Blankenship. They helped me understand the intricacies of industrial research and laid the foundation for my professional journey.

A special thanks is due to my friends who made my graduate years so memorable. Another special mention to Dr. Partha De, for being a friend, guide and philosopher during these five years and other senior UTD CS PhDs who successfully navigated the PhD journey and shared their experiences with me: Dr. Chiradeep Roy, Dr. Kinjal Basu, and Dr. Sagnik Dakshit: thank you for the late-night conversations and unwavering support. My heartfelt thanks also goes to my best friends Navanil Sengupta, Rohan Paul, Novonil Das, Abhigyan Sinha, and Pritom Das Radheshyam for all the insightful (and sometimes not-so-insightful) discussions that made these years unforgettable. I must also thank Princy Doshi, Swati Anwesha, Kongkona Borah, Amisha Srivastava, Souvik Das, Suprovo Ghosh, Antareep Gogoi and Pramit Ghosh for being the rock-solid support system that I could always count on.

My journey into academia began with the encouragement of many influential figures, such as my undergraduate advisor, the late Dr. Dick Blandford, as well as Dr. Katherine Chandler, Dr. Christina Howe, Dr. Robert Morse, and Mark Randall, who inspired me to pursue graduate studies and ultimately a doctorate. A special thanks to my highschool mentors who developed the virtue of hard work and perseverance: Sheila Huff (Principal), Kaye Pickett, Paula Boenigk (who taught me the value of critical thinking), Denise Dunlap, Amy Bonenberger, Michele Hays, Niti Moore, Melanie Preston, Andrew Weinzapfel, Michael Schauss, and Shane Burkhart.

I also want to acknowledge Dallas Bongs Cricket Club and AnkurDFW for creating a welcoming and supportive community for international students like myself. Being a part of these clubs allowed me to unwind, play my favorite sport (cricket) and stay connected to my Bengali roots, all while pursuing my academic dreams.

Finally, I am eternally grateful to my family. My parents, Kingsuk Mukherjee and Dr. Sharmistha Mukherjee, have been my guiding lights. Thank you to my brother, Spandan Mukherjee, and my extended family— (my paternal grandparents) the late Dr. Sisir Kumar and Anju Mukherjee, (my maternal grandparents) Dr. Sati Prasad and Subra Acharya, (my maternal aunts) Kanistha Acharya and Dr. Mongistha Acharya, (my paternal aunt) Dr. Kakoli Mukherjee, and my cousins, Arjun Chakraborty, and Bharsha Acharya—their love and encouragement means the world to me. Words cannot express how much they all mean to me. I wish my paternal grandfather, the late Dr. Sisir Kumar Mukherjee, could have been here to see this moment. From swimming across the Ganges as a Bangladeshi Hindu refugee in India to earning his doctorate, he laid the foundation for this achievement. I stand on the shoulders of such an enigma; we made it, and I hope this makes him proud. Last, a "thank you" to myself, Kunal Mukherjee, for persevering and not giving up—even when the situation seemed long and uncertain. Sometimes, all we can do is keep swimming.

November 2024

IOT INTEGRATION, ADVERSARIAL ATTACKS, AND THREAT EXPLANATIONS IN PROVENANCE-BASED INTRUSION DETECTION SYSTEMS

Kunal Mukherjee, PhD
The University of Texas at Dallas, 2025

Supervising Professors: Kangkook Jee, Chair
Murat Kantarcioğlu, Co-Chair

System provenance analysis has become the predominant approach for defending against sophisticated attackers. System provenance analysis captures causal and informational flow dependencies by correlating telemetry data across key system resources such as processes, files, and network sockets. These dependencies are efficiently represented as *system provenance graphs*, which are directed, heterogeneous, and multi-attributed. These system provenance graphs can be used by *Provenance-based Intrusion Detection System*s (PIDSs) to train *adaptive behavioral* Machine Learning (ML) models for intrusion detection tasks. PIDSs can effectively thwart Advanced Persistent Threat (APT) actors and Fileless Malware writers since they can measure the program behavioral deviations. Graph Neural Networks (GNNs) are the *de-facto* standard for learning from graphs. Consequently, GNN-based PIDS can detect zero-day and mimicry attacks by measuring deviations in program behavior.

Despite their undeniable advantages, modern PIDSs still face several open problems: *(1)* current system provenance analysis techniques are designed primarily for resource-rich environments, leaving IoT ecosystems vulnerable; *(2)* the resilience of PIDS against dedicated adversaries have not been fully examined; *(3)* GNN-based PIDS operate as black-box models, lacking transparency in their detection decisions.

This dissertation addresses these three key challenges in system provenance analysis: extending provenance analysis to IoT environments, improving robustness against adversarial attacks, and enhancing the explainability of GNN-based PIDS.

First, we introduce ProvIoT, a federated edge-cloud security framework that brings PIDSs to resource-constrained IoT devices. ProvIoT leverages federated learning to minimize network and computational overhead while maintaining high accuracy in detecting stealthy attacks, even in diverse real-world environments.

Next, we present ProvNinja, an adversarial testing framework designed to evaluate the robustness of PIDSs against realistic evasive attacks. ProvNinja generates adversarial attack variants that closely mimic benign system behaviors, allowing it to effectively test the resilience of State-of-The-Art (SOTA) PIDSs. Our experiments reveal vulnerabilities in current security models, leading to reduced detection rates in realistic attack scenarios.

Finally, we develop ProvExplainer, an explainability framework for GNN-based PIDSs to provide interpretable, security-focused explanations. ProvExplainer projects the GNN's decision boundaries onto the interpretable surrogate model's feature space (*e.g.,* discriminative subgraph patterns). By integrating with SOTA GNN explainers, ProvExplainer improves both precision and recall in explaining stealthy attacks (*i.e.,* APTs campaigns and Fileless malware) detections, offering a transparent and verifiable tool for security operations.

Together, these contributions offer scalable, robust, and explainable security solutions for increasingly interconnected and vulnerable digital infrastructure.

TABLE OF CONTENTS

ACKNOWLEDGMENTS v

ABSTRACT viii

LIST OF FIGURES xiv

LIST OF TABLES xvi

CHAPTER 1 INTRODUCTION 1

- 1.1 Overview 1
- 1.2 Dissertation Outline 4

CHAPTER 2 BACKGROUND 6

- 2.1 System Provenance Overview 6
- 2.2 System Provenance Data Schema 7
- 2.3 System Provenance Graph 7
- 2.4 System Provenance-based Intrusion Detection System (PIDS) 8
- 2.5 Graph Neural Network (GNN)-based PIDS 11
- 2.6 Explainable ML in Security 11
- 2.7 Ground-truth Verification in System Provenance 12
- 2.8 Fileless Malware 13
- 2.9 Advanced Persistent Threat (APT) Campaigns 17

CHAPTER 3 PRIVACY PRESERVING FEDERATED PIDS FOR IOT – PROVIOT 20

- 3.1 Problem Statement 20
- 3.2 Threat Model 20
- 3.3 ProvIoT Overview 21
  - 3.3.1 Federated Architecture: Local Brain 22
  - 3.3.2 Federated Architecture: Cloud Brain 24
- 3.4 Federated Detection in IoT Domain 25
  - 3.4.1 Provenance Graph Building and Subgraph (Path) Selection 26
  - 3.4.2 Document-to-Neural Embedding Model 28
  - 3.4.3 Federated PIDS: AutoEncoder 28
- 3.5 Evaluation 29

3.5.1 Dataset 30
3.5.2 Experimental Protocol 35
3.5.3 Fileless Malware Detection 36
3.5.4 APT Campaign Detection 38
3.5.5 Federated Learning Benefits 40
3.5.6 ProvIoT Overhead 41
3.6 Related Works 42
CHAPTER 4 EVASIVE ATTACK GENERATION FRAMEWORK – PROVNINJA 45
4.1 Problem Statement 45
4.2 Threat Model 46
4.3 ProvNinja Overview 47
4.4 Program Profile Generation – Frequency History of Events 48
4.5 Identifying Conspicuous (or Rare) Events 49
4.6 Feature Space Evasion 50
4.6.1 Evasive System Events (or Gadget) Finder 52
4.6.2 Applying Gadget Chains 52
4.6.3 Camouflaging Gadgets 53
4.7 Problem Space Evasion 56
4.7.1 Problem Space Constraints 57
4.7.2 System Provenance Filter Rules 58
4.8 Evaluation 61
4.8.1 Evaluation Methodology 62
4.8.2 Evaluation Datasets 63
4.8.3 Dataset Statistics 66
4.8.4 Baseline Performance of ML Detectors 68
4.8.5 Feature Space Evasion 69
4.8.6 White-box and Blind Threat Models 72
4.8.7 Problem Space Realization 73
4.8.8 Surrogate Dataset Effectiveness 77

4.8.9 Transferability Evaluation . . . 78
4.9 Related Works . . . 79
CHAPTER 5 EXPLAINING GNN-BASED PIDS – PROVEXPLAINER . . . 81
5.1 Problem Statement . . . 81
5.2 Threat Model . . . 81
5.3 ProvExplainer Overview . . . 82
5.4 Graph Structural Features . . . 84
5.4.1 Initial Compromise . . . 86
5.4.2 Establishing a Foothold . . . 86
5.4.3 Deepen Access . . . 87
5.4.4 Lateral Movement . . . 87
5.4.5 Look, Learn, and Remain . . . 88
5.5 Creating Surrogate Decision Trees (DTs) using Graph Structural Features . . . 88
5.6 Interpreting GNN-based IDS Detections Using Surrogate DTs . . . 89
5.7 Combining SOTA GNN Explanation Methods with ProvExplainer . . . 90
5.8 Evaluation . . . 91
5.8.1 Evaluation Protocols . . . 91
5.8.2 Evaluation Tasks . . . 93
5.8.3 Evaluation Datasets . . . 95
5.8.4 Dataset Statistics . . . 97
5.8.5 Graph Structural Feature Evaluation . . . 101
5.8.6 Ablation Study . . . 103
5.8.7 ProvExplainer vs. SOTA Explainers . . . 107
5.9 Case Studies . . . 110
5.9.1 FiveDirections: Browser Extension . . . 111
5.9.2 FiveDirections: Copykatz . . . 113
5.9.3 Trace: Phishing E-mail . . . 116
CHAPTER 6 FUTURE WORK AND CONCLUSION . . . 119
6.1 Future Research Directions . . . 119

6.1.1 Real-Time Prevention 119

6.1.2 Automated Evasive Attack Generation 119

6.1.3 Defense Against Evasive Attacks 120

6.1.4 Environmental Dependencies of Evasive Attacks 120

6.1.5 Adversarial Manipulation of Graph Features 121

6.1.6 Support for Fine-Grained Detection Tasks 121

6.2 Conclusion 121

REFERENCES 124

BIOGRAPHICAL SKETCH 136

CURRICULUM VITAE

LIST OF FIGURES

2.1 FritzFrog malware impersonating `nginx` web-server. 15

2.2 Enterprise APT scenario. 18

2.3 APT attack on Docker supply chain. 19

3.1 The federated framework of ProvIoT. 21

3.2 The detection pipeline of the Local Brain. 25

3.3 Example causal paths extracted from a provenance graph, G_1, generated for process, P_1. Using the extracted causal paths the sentences are formed for a document, D_1. 27

3.4 Example detection workload for graph G_1 in Figure 3.3. After the document D_1 is formed, the causal sentences in the document are converted into feature vectors (fv) using doc2vec model. Then the fv are fed into the AutoEncoder to get the reconstructed fv. Sentences are flagged as anomalous if the mean squared error between the original fv and the reconstructed fv is above a threshold determined during training. 28

3.5 Attacker injects and creates Fileless malware as a child process of `motion` process. The provenance graph captures the attacker's behavior which can be used for detection. 34

3.6 High detection accuracy of ProvIoT against APT attacks using federated learning, some rare exceptions which are discussed in §3.5.4. 38

3.7 (a) Federated performance is similar to centralized performance on the same data. (b) Increasing the number of clients increases performance by increasing the amount of data in the system. 40

3.8 On RaspberryPi 4B, Local Brain's processing and prediction uses <10% CPU and 65MB memory. Model training takes about 375MB memory and <10% CPU. . . 41

4.1 ProvNinja framework. 47

4.2 Change in regularity score due to gadget usage. 52

4.3 Events added and F1 score vs gadget length. In each evaluation scenario, bars represent number of additional events whereas solid lines are for F-1 score trends. 71

4.4 Realization effort for larger graphs takes more time, but there is a diminishing result since the number of rare edges and gadgets are limited for a particular attack stage. 75

4.5 Attack transferability visualization. Each cell contains the F1 score of the defense model (columns) that is measured against the evasive attacks crafted against a target model (rows). Lower values (lighter colors) indicate better evasion. 78

5.1 ProvExplainer framework. 82

5.2 Structural Graph Features. Squares are processes and circles are files. *Write* edges are blue, *read* edges are green, *execute* edges are red, and *process creation* edges are orange. 85

5.3 Distribution of nine subgraphs (Table 5.1) in different datasets. *B* means benign and *A* means anomaly. 85

5.4 Representative graphs of the different categories for `python` (linux), `powershell.exe` and `firefox.exe`. 95

5.5 Effectiveness of graph model explainers at identifying documented entities (§5.8.1), measured using precision and recall. 107

5.6 Effectiveness of graph model explainers at identifying documented entities (§5.8.1), measured using precision and recall as more nodes are included in the explanation. ProvExplainer outperforms SOTA explainers on anomaly detection tasks and remains competitive in classification tasks. 109

5.7 FiveDirections: the attacker exploits the target via a malicious `Firefox` extension. 111

5.8 FiveDirections: the attacker gains C2 connections and installs `Copykatz` through a `Firefox` exploit. 114

5.9 Trace: after an employee clicks on a phishing link, `Firefox` installs multiple Trojans to exfiltrate sensitive data. 116

LIST OF TABLES

2.1 Node and egdge types for in-house system provenance with associated attributes. 7

2.2 Top 10 impersonation targets for Fileless malware. 16

3.1 The IoT applications chosen for evaluation as well as their usage examples. . . . 30

3.2 Number of vertices and edges used to create a benign profile for IoT applications and system programs 31

3.3 Number of vertices and edges used to create IoT Malware and APT attack profile 31

3.4 APT TTPs for cyber-killchain stages 32

3.5 ProvIoT is highly effective in distinguishing IoT malware impersonating as benign system process as evident from high F1 scores. Grey cells contain low F1 score to indicate indistinguishable malware behavior for system process, discussed in §3.5.3. 37

3.6 Federated attack scenarios result for FritzFrog attack 38

4.1 Example gadgets with their normalized regularity score and problem space rejection reason. Regularity scores are normalized from 0 to 10, with a high score indicating higher regularity. 54

4.2 APT attack stages first showing the original attack and then the attack using gadget chain along with their regularity score. 55

4.3 presents the detection results for the baseline, randomly perturbed, and Ninja attacks, with a lower F-1 score indicating better evasion. We replace rare edges with a random sequence of programs in random perturbations and with evasive gadgets in Ninja attacks. We display differences from the baseline values inside parentheses. 63

4.4 Benign graph size for system programs. 67

4.5 ProvNinja evasion for ShadeWatcher (Zengy et al., 2022). 70

4.6 ProvNinja's performance under White-box, Black-box and Blind threat model, evaluated for two configurations of Blind (ProvNinja) and Blind (random perturbation). 72

4.7 Number of unviable candidates removed by each recommendation §4.7.2. 74

4.8 The detection results of the attacks generated from the benign, surrogate, and random dataset (lower numbers indicate better evasion). Rare edges and the gadget chains are found using the data. The random data is generated by intermixing DARPA TC datasets. 77

5.1 Summary of program behavior patterns. 83

5.2 Datasets used for classification: `python`(linux), `powershell.exe` and, `firefox.exe`. 98

5.3 APT and Fileless Malware graph statistics. 98

5.4 Classification dataset graph statistics. 99

5.5 Surrogate Decision Trees (DTs) have high agreement with the decisions of GNN measured using the WMA F1 score. Grey cells low agreement with the GNN (discussed in §5.8.5). 101

5.6 Agreement of surrogate DTs with the GAT model across different feature subsets. The best feature subsets are highlighted. 103

5.7 WMA F1 of surrogate DTs approximating a GraphSAGE model across different feature subsets. The best feature subsets are bolded. 104

CHAPTER 1
INTRODUCTION

1.1 Overview

System provenance analysis has become a cornerstone of modern cybersecurity (*i.e.,* enterprise and supply-chain security), providing detailed tracking of system events to defend against increasingly sophisticated cyber threats (FireEye, 2020; Malwarebytes, 2022; Eddy, 2024; Telychko, 2024). System provenance (King and Chen, 2003a) captures causal and temporal relationships between system operations, such as process executions, file modifications, and network connections. These relationships are most effectively represented as graphs. These provenance graphs offer a rich, fine-grained view of system behaviors, making them invaluable for detecting and thwarting advanced cyber-attacks (*i.e.,* APT campaigns and Fileless malware). Provenance graphs enable *Intrusion Detection Systems (IDSs)* to effectively model dynamic program behaviors, making it easier to distinguish malicious activities and backtrack from the Point-Of-Interest (POI) of the attack to its source.

Provenance analysis transforms raw system event logs into directed, heterogenous multi-attributed graphs, where nodes represent system entities (*i.e.,* processes, files, and network sockets), and edges represent their interactions (*e.g.,* `READ`, `WRITE`, and `CREATE`). By analyzing patterns in these graphs, security systems can detect deviations from normal behavior, flagging them as potential threats. Unlike static detection mechanisms, which rely on signatures or heuristics, *Provenance-based Intrusion Detection Systems (PIDSs)* offer a more dynamic and comprehensive approach, identifying novel, stealthy attacks based on behavioral analysis. These capabilities are particularly important in noisy environments where traditional defenses fail, such as against long running APT campaigns or Fileless Malware masquerading as benign system programs.

Despite their promise, provenance-based solutions face three key challenges, including scalability in computationally limited IoT environments, resilience against adversarial manipulation, and interpretability of security alerts. The following works extend the domain of provenance analysis to address these challenges.

In ProvIoT, we explore the applicability of PIDS to the IoT domain, where resource constraints such as limited CPU, memory, and bandwidth make traditional provenance solutions impractical. IoT devices, ranging from wearables to autonomous vehicles, are now deployed at a massive scale, making them frequent targets of data theft, cryptomining, and denial of service (DoS). To bypass legacy security systems in IoT, attackers are increasingly adopting stealthy techniques such as Fileless malware or Living-of-the-land malware (Harpaz, 2020). While provenance graphs are powerful for tracking behaviors, processing them is computationally challenging on the limited hardware capacity typical of IoT devices.

To address this challenge, ProvIoT introduces a federated edge-cloud collaborative security architecture tailored for IoT environments. This system optimizes event collection and summarization using an in-memory database. It extracts critical subgraphs, or causal paths, from the full provenance graph, further reducing the overhead. ProvIoT's federated architecture protects each device's privacy by only sharing local models' weights across the network *e.g.,* system data never leaves the edge device. The local model weights are aggregated in the cloud using federated learning (McMahan et al., 2017) to create global behavioral models, which are shared with the edge devices for on-device detection. As a result, the edge devices collaboratively train a detection model with experiences from each device in the federation. Since detection is performed on-device, even if an attacker disconnects a device from the network, the local detection engine remains functional. This decentralized approach allows IoT systems to scale their security defenses without overwhelming local resources, providing reliable protection across a wide array of devices.

In ProvNinja we focus on the challenges posed by adversarial attacks against PIDS, where sophisticated adversaries manipulate system behaviors to evade detection while achieving malicious objectives. PIDS are vulnerable to adversarial attacks that combine traditional masquerading techniques with behavioral mimicry, allowing them to evade detection by making malicious actions appear benign. The adversarial attacks are not only limited to the feature space (ML model latent representation) but are realizable in the problem space (*e.g.,* real-world system actions). Such attacks therefore present a significant and meaningful challenge for securing systems against advanced adversaries.

ProvNinja systematically tests the robustness of PIDSs (*e.g.,* traditional PIDS (Wang et al., 2020; Han et al., 2021) and GNN-based PIDS) by measuring their robustness against realistic adversarial examples. This involves identifying conspicuous system events in the traditional attack chain, replacing them with similar common system events, and then realizing the modified attack in real system executions (Metasploit, 2021; MetasploitVenom, 2021). By exploiting the vulnerabilities of ML models to adversarial manipulation, ProvNinja highlights the need for more robust security defenses that can withstand both known and unknown attack strategies. Evaluations show that current PIDSs struggle against these evasive tactics, underscoring the importance of robustness in provenance security solutions.

While ML models built on provenance graphs can effectively detect complex attack patterns (adversarial attacks notwithstanding), their opaque nature can make it difficult for security analysts to understand and trust their decisions. The lack of explainability in these models, particularly GNNs, limits their adoption in security-critical environments where understanding the rationale behind a detection is just as important as the detection itself. Security practitioners require explanations that are not only accurate but also meaningful in the context of real-world threats.

ProvExplainer tackles this challenge by introducing an explainability framework that enhances GNN-based security models with *security-aware* graph-structural features. By using these features exclusively for interpretation, this approach decouples the decision-making process from feature engineering. This allows feature development to focus on providing security-relevant explanations rather than improving detection accuracy. ProvExplainer introduces interpretable surrogate models (*i.e.,* decision trees) that capture discriminative graph substructures linked to diverse malware behaviors (*e.g.,* drive-by downloads, system probing, and malware staging) and use these patterns to explain decisions made by the GNN-based PIDS model. This framework not only improves the interpretability of GNN-based detectors but also facilitates the integration of domain-specific knowledge into the explanation process, allowing security experts to make informed decisions. Evaluations demonstrate that ProvExplainer surpasses state-of-the-art (SOTA) explainers in providing security-aware explanations, making it a critical advancement in the adoption of ML for security.

Provenance-based analysis represents a powerful and flexible approach to securing modern computing environments, from IoT devices to enterprise networks. While provenance graphs provide rich and behavioral data for detecting and mitigating advanced attacks, several challenges must be addressed for widespread adoption: *e.g.,* scalability in constrained environments, resilience against adversarial manipulation, and the need for explainability. Through ProvIoT, ProvNinja, and ProvExplainer, we extend the capabilities of provenance-based security solutions, making them more scalable, robust, and interpretable for real-world deployment. These advancements lay the groundwork for future research and practical applications in securing complex and dynamic systems.

1.2 Dissertation Outline

In this section, we will discuss the layout of the remaining chapters and summarize them.

The background to understand this dissertaion is presented in Chapter 2. We give an overview of the provenance domain, then detail our system provenance data collection pipeline and data format. This is followed by a discussion of *Intrusion Detection System* (IDS) detection explanability and ground-truth verification. Then, we survey PIDS and the SOTA GNN-based IDS. Finally, we include a discussion of Fileless Malware and Stealthy APT attacks.

Chapter 3, “Privacy Preserving Federated PIDS for IoT Domain – ProvIoT”, presents the development and evaluation of ProvIoT, a provenance-based security detection system tailored for IoT environments. This chapter discusses how ProvIoT counters stealthy attacks using federated learning and on-device detection. It introduces a new design choice for federated edge-cloud collaborative security learning, which streamlines computationally expensive graph-based behavioral security within the IoT context. Additionally, we evaluate the effectiveness and efficiency of ProvIoT using realistic attack cases, including Fileless malware.

Chapter 4, “Evasive Attack Generation Framework – ProvNinja” presents a systematic study of adversarial evasion against provenance-based ML security detectors. The chapter explains how ProvNinja leverages a publicly available surrogate dataset to implement a data-driven approach that constructs evasive attack vectors while complying with realistic system constraints. It further evaluates against various PIDSs, using comprehensive datasets gathered from public and private real-world environments that include both benign and malicious data.

Chapter 5,“Explaining GNN-based PIDS – ProvExplainer” describes the explanation framework ProvExplainer, which aims to increase the transparency and verifiability of PIDSs. This chapter outlines how ProvExplainer examines graph structural features linked to system actions and security events, aided by security domain knowledge. It also highlights the development of security-aware features that enable surrogate models to achieve high agreement with the detection model when detecting APT and Fileless malware. In our evaluation, we show that integrating ProvExplainer with SOTA GNN explainers (*e.g.,* GNNExplainer, PGExplainer, and SubgraphX) improve ground-truth explanations.

Finally, Chapter 6 concludes the dissertaion by illustrating these works’ limitations and presenting some avenues for future research.

CHAPTER 2

BACKGROUND

In this section, we provide an overview of the system provenance domain, exploring its downstream applications of IDS and GNN-based IDS. We then discuss the role of explainability in security contexts, highlighting its connection to ground truth verification. Lastly, we offer background on stealthy attack techniques, including Fileless malware and APT attack campaigns.

2.1 System Provenance Overview

System provenance analysis (Liu, Zhang, Li, Jee, Li, Wu, Rhee, and Mittal, Liu et al.), originally proposed by King et al. (King and Chen, 2003a) for host-based system monitoring. System provenance operates through the installation of a data collection agent on each host to collect `syscall` level system events. These events are then sent to an in-memory or external database to build a causality graph by associating data and control dependencies between processes, files, and network resources. The events that system provenance collects are as follows: (1) process events, such as process `CREATE`; (2) file events, including file `READ`, `WRITE`, and `EXECUTE`; and (3) network events, including socket `CREATE`, `READ`, and `WRITE`.

With the increased deployment of provenance-based security solutions in the last decade (CrowdStrkie, 2020), the output of system provenance, the *provenance graph*, forms the foundation for graph-based learning and detection approaches. In this regard, provenance graphs best represent the runtime characteristics of system entities and have quickly become an essential source of input to model a program's runtime behavior. Along with recent developments in graph-based learning approaches (Kipf and Welling, 2016; Pan et al., 2019), research on behavioral modeling and its application for anomaly detection has gained considerable momentum (Hassan et al., 2019; Wang et al., 2020; Rehman et al., 2024; Jia et al., 2024; Cheng et al., 2024).

Table 2.1: Node and egdge types for in-house system provenance with associated attributes.

	Types	Attributes
Nodes (resources)	process	signature, executable name, pid
	file	owner (uid, gid), name, inode
	socket (IP)	dstip, srcip, dstport, srcport, type
Edges (events)	process → process	command args, starttime
	process → file	read, write, amount
	process → IP address	send, recv, amount

2.2 System Provenance Data Schema

System provenance analysis (King and Chen, 2003a; Liu, Zhang, Li, Jee, Li, Wu, Rhee, and Mittal, Liu et al.; Wang et al., 2020; Hassan et al., 2019) leverages data collection agents on end-hosts to collect interaction events among key system resources: processes, files, and network sockets. This work relies on in-house data from 21 hosts in a university, the DARPA Transparent Computing (TC) dataset (DARPA, 2019). Our in-house data collection accumulates 13 GB to 92 GB daily, tracking around 875 unique programs, 7,025K processes, 4,824K network connections, and 111,583K file operations.

Our system provenance data schema, detailed in Table 2.1, is similar to DARPA's Common Data Model (CDM) (DARPA, 2019) schema, but we omit memory objects, registry events, and thread distinctions within a process. These choices were made to balance real-world overhead constraints of load balancing and storage. We also established a malicious testbed to generate malware execution traces. To ensure the freshness and realism of our malware samples, we utilize Cyber Threat Intelligence (CTI) feeds (ATT&CK®, 2022b), the VirusTotal API (VirusTotal, 2021), and penetration testing tactics, techniques, and procedures (TTPs) (Metasploit, 2021; Strike, 2023).

2.3 System Provenance Graph

System provenance traces information, control, and temporal dependencies of a computer system (King and Chen, 2003b; Liu, Zhang, Li, Jee, Li, Wu, Rhee, and Mittal, Liu et al.). By examining system-call logs, we can monitor the behavior of all processes on a system, tracking all `READ`, `WRITE`, and `EXECUTE` operations on files and network sockets. Please note that we refer `sockets` to indicate IP-based network connections. We create a *provenance graph* by associating the casual dependencies between these system events. Formally, a provenance graph is a connected set of timestamped edges $e = (u, v, r)$ where $u, v \in \{processes \cup files \cup sockets\}$ and u is causally dependent on v (*e.g.,* a file u is written to by a process v), and r is the relationship between the nodes (*e.g.,* `READ` and `WRITE` files, and `EXECUTE` programs, `SEND` to and `RECEIVE` from sockets).

The provenance graph, annotated with various attributes for nodes (*e.g.,* resources, such as processes, files, and network sockets) and edges (*e.g.,* system calls over resources), includes process executable names, IP addresses, access types, and more. These graphs serve as invaluable forensic analysis tools, helping discover points of entry, tracing lateral movement, and assessing the scale of damage. However, the fine-grained nature of these graphs leads to high complexity and heterogeneity, causing their size to grow exponentially over time. Consequently, researchers actively explore various approaches to reduce both analysis and storage overheads (Fei et al., 2021; Tang et al., 2018; Xu et al., 2016).

2.4 System Provenance-based Intrusion Detection System (PIDS)

PIDS have focused on stealthy attacks such as APT campaigns to address known limitations of the traditional security defenses. The signature-based IDS is built on static resources (*e.g.,* filename, IP, domain names) and their derived artifacts (*e.g.,* hashes) which can trivially be forged by legitimate system operations or advanced attacker techniques (O'Kane et al., 2011; Song et al., 2007) or Fileless techniques (Song et al., 2019; Barr-Smith et al., 2021), which have become a de facto attack vector, allowing attackers to impersonate their identity into the system's normal behavior.

The rule-based IDS (yar, 2020; sno, 2021; sur, 2020) extends signature-based IDS by encoding behavioral patterns around the highly-sensitive system resources. However, this is also limited in detecting stealthy attack campaigns that take place over a long duration involving dependencies over multiple resources. The signature-based IDS relies on static resources (e.g., filenames, IPs, domain names) and their derived artifacts (e.g., hashes), which can be easily evaded by advanced attackers who can incorporate benign looking system operations or stealthy attack techniques (O'Kane et al., 2011; Song et al., 2007, 2019; Barr-Smith et al., 2021). These methods have become a de facto attack vector, allowing attackers to blend their identity into the system's normal behavior.

The rule-based IDS (yar, 2020; sno, 2021; sur, 2020) expand upon the signature-based IDS by encoding behavioral patterns around highly-sensitive system resources. However, this approach also has limitations in detecting stealthy attack campaigns that take place over a long duration. This can also be bypassed by Fileless techniques (Song et al., 2019; Barr-Smith et al., 2021), which have become a de facto attack vector, allowing attackers to impersonate their identify into the system's normal behavior.

For instance, using the Fileless technique, the attacker can access to highly sensitive resources (*e.g.,* `/etc/passwd` or `/etc/shadow`) by impersonating one of many legitimate programs (*e.g.,* `/usr/bin/sshd`, `/bin/passwd`, `/bin/ls`) [1] Recent advancements in ML research have extended anomaly detection to provenance graphs (Zengy et al., 2022; Rehman et al., 2024; Cheng et al., 2024), allowing individual processes to be monitored for unusual behavior at runtime.

Graphs are known for their ability to capture complex relationships between nodes and edges, which are translated to system resources (*e.g.,* file, process, and sockets) and causal dependencies among them. Structural relationships captured by system provenance offer

---

[1] From event database history, we found on average 53 and 10 programs which access `/etc/passwd` and`/etc/shadow` respectively over two weeks period.

robust features which are hard for an adversary to manipulate. Unlike resource names or hashes, it would take a larger effort to manipulate a long list of structural dependencies among system resources while seen benign to anomaly detection models. Therefore, despite its data collection and modeling costs, the PIDS has become a promising countermeasure against stealthy attacks and APT campaigns.

Graph-based analysis and its application for anomaly detection are computationally expensive and require a large amount of training data. Hence, the research community has introduced several approaches to embed provenance graphs into a vector space to train ML models (Han et al., 2020; Wang et al., 2020; Hassan et al., 2019; Han et al., 2021). Path-based models extract graph subcomponents (*e.g.,* causal paths) from the provenance graph and vectorize them to leverage existing learning approaches (Breunig et al., 2000; Google, 2021b). Although efficient even against large volume of graph inputs, the embedding approach compromises the detection accuracy by sampling subset of provenance graphs, losing the context of the entire graph. Recent advancement of framework support and associated technologies (DGL, 2022), we can leverage Graph-Neural Network (GNN) (Chaudhary et al., 2019) techniques to digest the entire provenance graph directly.

In this dissertation, *ML detectors* refer to learning-based security detectors that operate on system provenance graphs encompassing path-based and graph-based models. The path-based models first deconstruct the graph into path embeddings and train on them, whereas Graph-based models work on entire graphs (rather than paths). We specifically discuss *(1)* two path-based models — ProvDetector (Wang et al., 2020), which uses Local Outlier Factor (LOF) on path embeddings to find outliers; and SIGL (Han et al., 2021), which uses an AutoEncoder (AE) to identify anomalous paths by characterizing the abnormality with the reconstruction loss of the path embeddings extracted from the AE model, *(2)* two Graph-based models — a GAT model named GAT ("Structure-Based Graph Attention Network") that uses the full provenance graph without node and edge attributes to distinguish benign and anomalous graphs, relying only on the structure of the graphs; and another similar model that includes the node and edge attributes along with graph structural features, which we named as Prov-GAT ("Attribute-Based Graph Attention Network").

While features and attributes for individual nodes and edges are local and easily manipulated, the structural relationships among them would pose difficulties for the attacker as it would require a series of complex operations to make graph-level changes and still be seen as benign by the anomaly detection models. To demonstrate ProvNinja's generality, our research implements evasive attacks against all of these provenance-based ML models.

2.5 Graph Neural Network (GNN)-based PIDS

Our research employs an industry-standard GNN framework to model and explain system provenance graphs, leveraging DGL's mature development ecosystem (DGL, 2022) for alignment with current analytical techniques and streamlined integration into real-world applications. Despite the security community's historical preference for custom detectors[2] (Yang et al., 2023; Zengy et al., 2022; Cheng et al., 2024; Rehman et al., 2024; Goyal et al., 2024) due to the complexity and heterogeneity of provenance graphs, we chose a general GNN framework for long-term integration benefits and broader impact. Notably, the DGL community has integrated our heterogeneous GNN enhancement.

2.6 Explainable ML in Security

Explainability of GNNs

Recent research in GNN explainers (Ying et al., 2019; Luo et al., 2020; Yuan et al., 2021) has advanced in identifying key nodes, edges, or subgraphs in GNNs, and are categorized into white-box and black-box explainers. White-box methods, *e.g.,* GNNExplainer (Ying

---

[2] It has only been recently that GNNs have started gaining attention within the community. Certain provenance-based ML detectors were even proposed before GNN frameworks became popular.

et al., 2019) and PGExplainer (Luo et al., 2020), access GNN internals, including model weights and gradients. Conversely, black-box methods like SubgraphX (Yuan et al., 2021) operate on model inputs and outputs, reducing coupling between the explanation framework and model architecture.

Explainability in Security Context

Due to the importance of explainability in the security domain, several explainers have been proposed for ML-based security analysis. LEMNA (Guo et al., 2018) focused on classifying PDF malware and detecting a function's entry point in binary code using regression mixture models as a localized surrogate to approximate the classifier's decision boundary.

Recent works such as CFGExplainer (Herath et al., 2022) and FCGAT (Someya et al., 2023) use deep surrogate models to explain GNN-based malware detection using control flow graphs or function call graphs. These methods exclusively work on homogeneous graphs, thus cannot be directly applied to provenance domain. Jacob et al. (Jacobs et al., 2022) proposed TRUSTEE, a framework that generates DT-based interpretations for ML models to detect shortcut learning (*e.g.,* problem underspecification). Their success clarifying model decisions about network packets inspired us to generalize the approach to the system provenance domain. The core challenge is that the system provenance domain relies on highly heterogeneous graph datasets, which are not natively consumed by DTs. By leveraging security-oriented graph structural features and cooperating with general-purpose GNN explainers, ProvExplainer enables DT-based explanations in the provenance domain.

2.7 Ground-truth Verification in System Provenance

In system provenance, "ground truth" refers to the real-world information against which the validity of a model's predictions are checked. In this dissertation, we approximate the ground truth using documentation created by security vendors, tech reports, and previous studies. The relevant processes, files, and network sockets mentioned in the documentation are designated as *documented entities*. We understand that the documentation provided by the security vendors can have experimental errors, as well as selection and experimental bias. To mitigate these problems, we aggregated information from multiple security vendors.

We extract *documented entities* with three methods: *(1)* referring to pre-existing malware profile databases that contain information from different security vendors, such as `VirusTotal` (VirusTotal, 2021), we obtain activity summaries detailing network communications, file system actions, and process behaviors; *(2)* we extract key entities (*i.e.,* process involved, files created and connections made) from tech reports (dar, 2023a,b) and system manuals (die.net, 2017); *(3)* we reviewed dataset documentation to identify the components of each attack present in the datasets.

Ground truth verification of ML model decisions in security-critical tasks has garnered significant attention, underscoring the role of explanations in unveiling the truth (Warnecke et al., 2019, 2020; Ganz et al., 2023). These studies highlight the effectiveness of white-box techniques in malware detection and vulnerability discovery, with further advancements in reconstructing ground truth around local explanations (Ganz et al., 2023). (Kosan et al., 2023) has shown high variance in explanations provided by traditional GNN explainers, raising reliability and applicability concerns.

2.8 Fileless Malware

Fileless Malware for IoT Devices

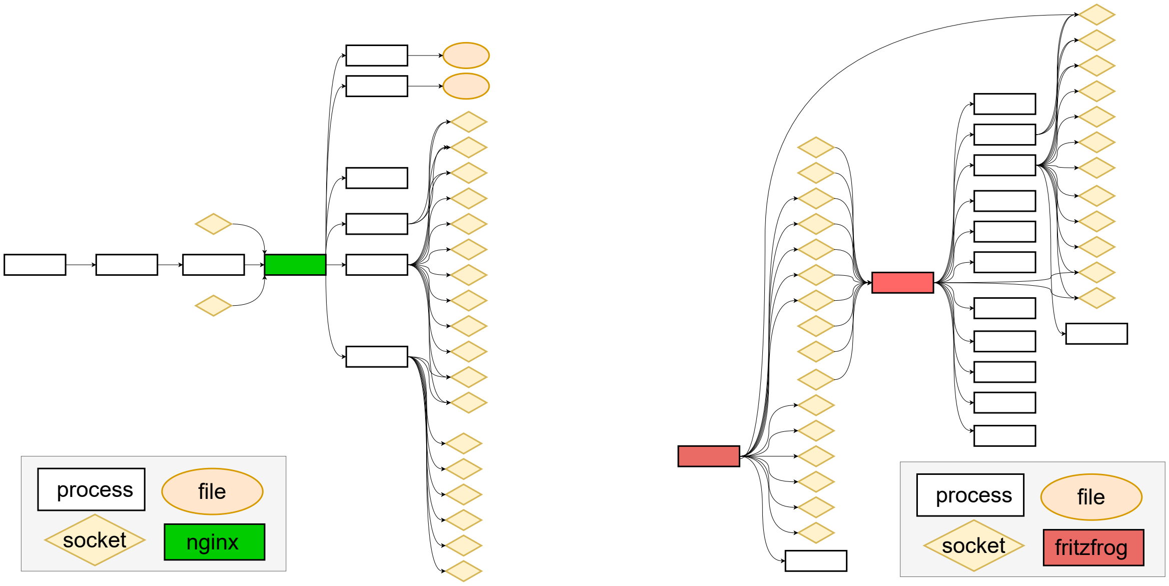
In this dissertation, *Fileless attack* refers to a group of attack techniques with no footprint in the file system. Alternative terms used in the field include “zero-footprint”, or “living off the land” (Barr-Smith et al., 2021).

Fileless malware are characterized by the impersonation of trusted off-the-shelf applications and pre-installed system utilities. Since many of these trusted applications are commonly used by users and system administrators, it is harder for defenses to block access to them to prevent such attacks completely. Such impersonation techniques have seen rising popularity in recent cyberattacks (Li et al., 2021; Barr-Smith et al., 2021). Instead of storing the malware payload directly onto the disk before executing it, this malware uses the strategy of "living off the land" by injecting it into benign running processes (*e.g.,* trusted applications) and avoiding detection by executing only in process memory. During runtime, the malware may also rename itself to a seemingly benign process name using a `prctl(PR_SET_NAME)` call. These impersonation approaches have diverged and evolved in multiple ways in IoT systems (Cozzi et al., 2018a; Costin and Zaddach, 2019). Some possible impersonation approaches are highlighted below.

Process Injection. `ptrace()` is a system API used to support code injection to another process for development purposes. However, attackers have abused `ptrace()` to inject malicious code into the memory of legitimate processes (ATT&CK®, 2021f).

In-memory Execution. The `memfd_create()` system API family creates an anonymous file in memory-mounted file systems. Using `memfd_create()`, an attacker can directly load malware from the memory space without writing a payload to the filesystem. This attack enhances the traditional attack strategy of storing malware in transient storage (*e.g.,* `/tmp`, `/var/run`, `/dev/shm`). With `memfd_create()`, the malware further reduces its footprint, preventing users from locating it with standard filesystem access even during runtime. Multiple loader frameworks (Cybersecurity, 2021) exist that are able to encode regular file-based malware into different Fileless variants.

Case Study: FritzFrog. In January 2020, a security group discovered and reported FritzFrog (Harpaz, 2020), a sophisticated peer-to-peer (P2P) malware botnet. FritzFrog is a crypto mining worm that breaks into and spreads through SSH servers. Written in Golang to natively target different architectures, FritzFrog uses Fileless techniques to leave no traces on the filesystems of the infected devices. We specifically consider FritzFrog in the context of IoT devices.



(a) Benign `nginx` (b) Impersonated `nginx`

Figure 2.1: FritzFrog malware impersonating `nginx` web-server.

FritzFrog performs file operations in memory to impersonate a regular benign system process's identity. After the initial break-in, FritzFrog masquerades as the `nginx` web server or the `ifconfig` process. The infected IoT device connects to a command and control (C&C) server via encrypted sessions to seemingly benign beacons. Then, the malware infects other IoT devices to mine cryptocurrencies by exploiting a weakness in SSH services. Figure 2.1 compares the behavior of the original `nginx` process and that of FritzFrog impersonating `nginx`. Although FritzFrog leaves no filesystem footprint, provenance-based intrusion detection systems can detect and defend against it as the behavior of benign `nginx` and FritzFrog are distinct.

FritzFrog is completely proprietary; its P2P implementation and C&C communication were entirely written from scratch, indicating that the attackers were high-profile security professionals. We obtained samples of FritzFrog and reversed the malware to trigger its Fileless behavior. We reproduced the Fileless behavior through network interceptions and

Table 2.2: Top 10 impersonation targets for Fileless malware.

Impersonating Programs	Malware Samples	Percentage of whole
`conhost.exe`	847	15%
`rundll32.exe`	821	14%
`python.exe`	822	13%
`svchost.exe`	734	12%
`explorer.exe`	673	11%
`reg.exe`	537	9%
`cscript.exe`	442	8%
`wmic.exe`	439	7%
`schtasks.exe`	329	6%
`nslookup.exe`	281	5%

specific command line inputs. In Section 3.5.3, we evaluate ProvIoT's effectiveness by comparing the captured behavior of FritzFrog against the model we built using execution traces of `Nginx` and `ifconfig` collected from our benign deployment.

Fileless Malware for Non-IoT Devices

Various tactics, techniques, and procedures (TTPs) are developed and shared to empower advanced attackers, which have contributed to the recent proliferation of major cybersecurity incidents. Fileless malware is one of the most noteworthy among these and regarded as a de facto attack vector for APT campaigns. Because Fileless malware does not write an executable to the file system, common threat detection schemes that scan the file system for suspicious artifacts are ineffective, allowing attackers to impersonate or inject behavior into common system programs. Fileless techniques are widely used in APT campaigns to hide malicious activities during lateral movement or to reduce the attack footprint of standalone malware(Song et al., 2019; Forrest, 2017; Kaspersky, 2020). Referring to the latest research (Barr-Smith et al., 2021), we established a large-scale Fileless malware dataset (refer to Table 2.2) for our research.

2.9 Advanced Persistent Threat (APT) Campaigns

APT campaigns exhibit two main characteristics: *(1)* a long-lasting nature, particularly during the lateral stage, and *(2)* the use of stealthy attack vectors to minimize the attacker's footprint and remain undetected throughout the campaign. While advanced security solutions have focused on tracing these attacks by mitigating system events with high-security implications, our study concentrates on the robustness of provenance-based ML detectors against evasion attempts by advanced adversaries. To adequately evaluate the provenance-based IDS approaches and their robustness, we implemented two realistic APT scenarios — Enterprise APT and Supply Chain Attack, alongside a large dataset for Fileless malware.

Unlike conventional attack vectors, stealthy attacks are designed to evade security detection systems by hiding malicious activities. Hence, advanced adversaries commonly use various Tactics Techniques and Procedures (TTPs) to forge static and easy to manipulate artifacts. They are advanced and skilled enough to craft their attacks to remain undetected by conventional security systems, which motivated the research community to develop advanced security solutions. Therefore, provenance-based IDS has become a promising defense to counter stealthy attacks and APT campaigns. While features and attributes for individual nodes and edges are local and easily manipulated, the structural relationships among them would pose difficulties for the attacker as it would require a series of complex operations to make graph-level changes and still be seen as benign by the anomaly detection models.

Enterprise APT. The phishing email attack as shown in Figure 2.2 can be classified according to MITRE ATT&CK framework into five major TTPs: Initial Access (ATT&CK®, 2022a), Establishing a Foothold (ATT&CK®, 2021a), Privilege Escalation (ATT&CK®, 2021b), Deepen Access (ATT&CK®, 2021e), and Exfiltration (ATT&CK®, 2021c). For our experiment, we were able to conduct the five TTPs using the well-known penetration testing framework (Metasploit, 2021; MetasploitVenom, 2021). The attack involves an attacker crafting a malicious macro (*e.g.,* malware named `java.exe`) embedded attachment

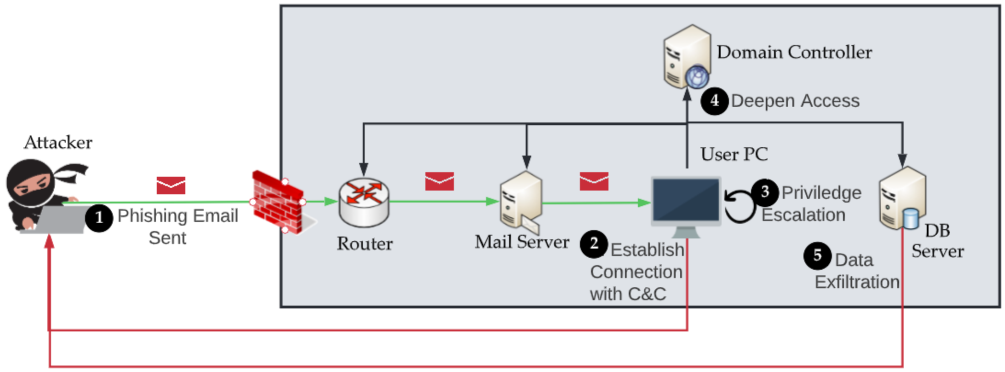


Figure 2.2: Enterprise APT scenario.

(*e.g.,* Excel document) which is sent to a machine victim through email that is inside an enterprise environment, as shown in Figure 2.2. The first TTP, Initial Access, is realized when the victim downloads and opens the email attachment.

The malicious macro starts a new malware process called `java.exe` which opens an initial connection with the attacker's command and control center (C&C) using port 443. The second TTP, Establishing a Foothold, is realized here. The attacker then performs Privilege Escalation by exploiting a vulnerability in (`notepad.exe`) (CVE, 2022). The attacker can then open a privileged command prompt (*e.g.,* `cmd.exe`).

Using the privileged command prompt, the attacker scans the network and breaches the LDAP server using port 445 to steal SQL database credentials. The attacker then runs specialized software (ATT&CK®, 2021d) to get the password hashes and LSA secrets. The fourth TTP, Deepen Access, is realized here as the attacker tries to penetrate the enterprise organization and infect more victims.

Once the SQL server is located, the attacker executes a malicious visual basic script file using `cscript.exe` to create another malware instance. This malware process executes SQL commands in the privileged shell using `osql.exe` as well as `sqlservr.exe` and then dumps out SQL DB data to the target's machine using stolen credentials. Finally, the attacker downloads the database dumps generated by the command and removes itself by deleting any temporary files, processes or executables created, completing the fifth TTP, Exfiltration.

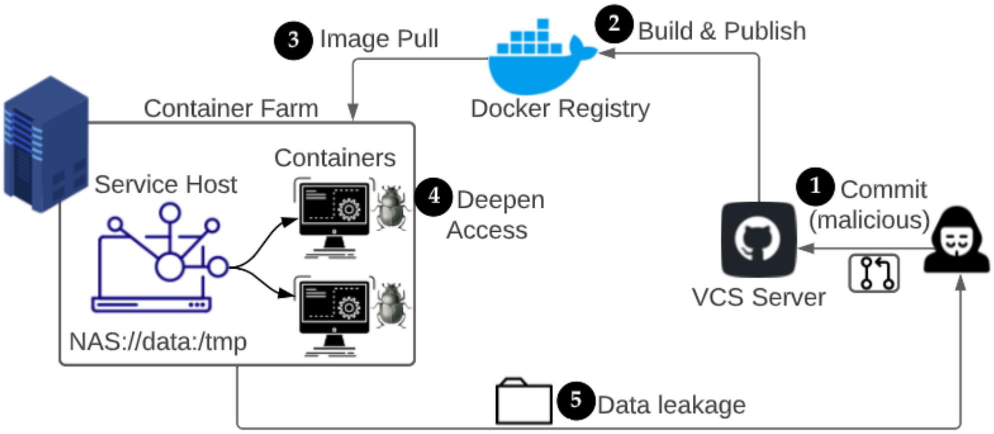


Figure 2.3: APT attack on Docker supply chain.

Supply Chain APT. The supply chain attack Figure 2.3 also contains the five TTPs mentioned in APT scenario one, but in three stages. It starts with the attacker committing a malicious docker image to a public repository which contains malicious changes to the docker compose file. The malicious docker image contains custom programs that allow the attacker to perform arbitrary file interactions after the docker compose file mounts unauthorized system directories. The exploitable docker image is deployed across the network using the victim's internal infrastructure.

When a victim pulls the malicious docker image they unknowingly complete the first TTP, Initial Access. Then, the victim runs it by giving the docker image privilege permission (*e.g.,* `sudo`), completing the second and third TTPs, Establishing a Foothold and Privilege Escalation. The modified docker compose file first mounts unauthorized directories and reads the contents of the home directories of the victim as well as any other user on the compromised machine. The fourth TTP, Deepen Access, is realized here as the attacker is able to penetrate different user's directories without their explicit permission or knowledge. Data is then exfiltrated by utilizing system programs that are popularly used such as `curl`, `wget`, completing the fifth TTP of exfiltration.

CHAPTER 3

PRIVACY PRESERVING FEDERATED PIDS FOR IOT – PROVIOT

3.1 Problem Statement

ProvIoT attempts to detect malicious behavior in IoT systems by learning the distribution of expected benign behaviors and reporting significant deviations from that expectation. We primarily consider APT scenarios (Mukherjee et al., 2023) and Fileless malware (mirai, 2016; vpnfilter, 2018; Harpaz, 2020) that impersonates one or more of a set of whitelisted programs to evade traditional IDS (Han et al., 2021) mechanisms. Similar to previous IoT security research (Ding, 2017; Cozzi et al., 2018a), ProvIoT aims to protect IoT devices which have been the primary target for stealthy adversaries (mirai, 2016; vpnfilter, 2018; Harpaz, 2020). Besides these devices, ProvIoT is also suited for IoT connected to microcontroller (MCU) based device families that interact with physical mechanisms for control and monitoring purposes.

3.2 Threat Model

Our threat model assumes that the data collection and summarization pipeline on the IoT device is trusted *i.e.,* the integrity of the provenance records are guaranteed by existing secure provenance systems (Wang et al., 2020; Han et al., 2021; Hassan et al., 2019; Mukherjee et al., 2023). This assumption is consistent with existing provenance research that requires end-host data collection and reporting (Hassan et al., 2019; Liu, Zhang, Li, Jee, Li, Wu, Rhee, and Mittal, Liu et al.; Wang et al., 2020). Securing and verifying the trustworthiness of the end-host data reporting is an important research topic that is orthogonal to our research (Ahmad et al., 2022). Procedural dataset poisoning is outside the scope of our work. We consider the use of distributed consensus protocols (Lamport et al., 1982) or attestation approaches that extend the root of trust with hardware level support (trustzone, 2018).

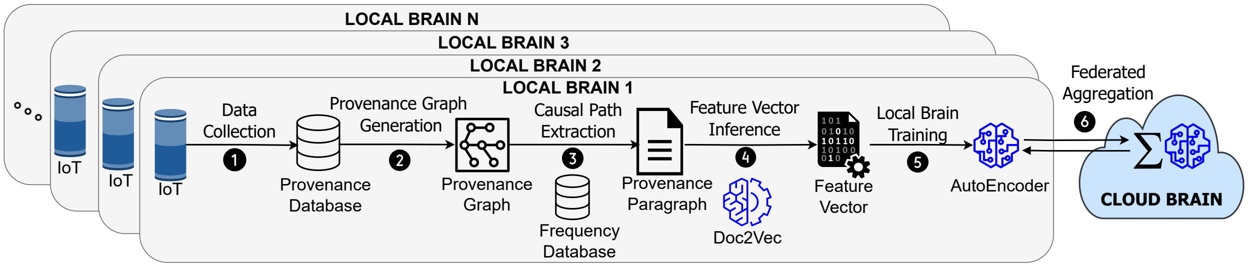


Figure 3.1: The federated framework of ProvIoT.

Attacks targeting the IoT platform, communication infrastructure (Acar et al., 2020), or the analysis process running in the cloud are outside the scope of this dissertation. We further assume that the reporting agents are honest and restrict our target IoT devices to those with at least 375MB of RAM (Mothukuri et al., 2021; Shahid et al., 2021) to support provenance summarization. Many modern commodity IoT devices (*e.g.,* smart thermostats (nes, 2015), smart watches (app, 2015), smart fridges (sam, 2022), smart doorbells (ama, 2020), and smart home devices (goo, 2017)) are equipped with 512MB or more of RAM.

3.3 ProvIoT Overview

Figure 3.1 presents the architecture of ProvIoT, that is composed of two collaborating subsystems: *Local Brains* and a *Cloud Brain.* Each Local Brain gathers host-level monitoring data from the IoT device into an in-memory database. It then summarizes the data and converts it to neural embeddings for ML model training. Data summarization only incurs 10% CPU usage and 65MB of RAM overhead. We can set the relevant local events and model training to run infrequently during low-load periods. After the local training, the Local Brain sends the updated neural weights to the Cloud Brain.

The Cloud Brain uses federated averaging (McMahan et al., 2017) to combine the weights received from the Local Brains into a global model, which is sent back to each Local Brain for use in detection. The Local Brain can then perform detection directly on the IoT device using the federated global model. Periodically, the Local Brain will synchronize with the

Cloud Brain, pushing up its local weights and fetching the updated global model. The *only* communication that the Local Brains have with the Cloud Brain is the communication of model weights during training. The Local Brains are fully capable of defending the IoT devices even when disconnected from the network.

To the best of our knowledge, ProvIoT is the first proposed provenance-based security detection approach in the context of IoT that counters stealthy attacks using federated learning and on-device detection. ProvIoT introduces a novel design choice for federated edge-cloud collaborative security learning by optimizing computationally expensive graph-based behavioral security mechanisms for IoT environments. We have extensively evaluated the efficiency and effectiveness of ProvIoT through realistic deployments, including adversarial scenarios carefully crafted with real-world attack cases and Fileless malware samples. To benefit the community and facilitate future research, we will make our dataset publicly available [1] and offer data collection support to researchers and practitioners.

3.3.1 Federated Architecture: Local Brain

We deploy a Local Brain to each IoT device to collect host-level monitoring data including process creations, file operations and network socket interactions. The Local Brain's training has the following major steps: *(1)* data collection, *(2)* provenance graph generation, *(3)* causal path extraction, *(4)* feature vector inference and *(5)* model training.

The first step in doing provenance analysis in IoT is data collection ❶, where we collect system monitoring data and create system event records. Similar to (Wang et al., 2020; Hassan et al., 2019; Liu, Zhang, Li, Jee, Li, Wu, Rhee, and Mittal, Liu et al.; Han et al., 2021; Mukherjee et al., 2023), we collect monitoring data for the following types of system entities: processes, files, and Unix domain sockets. Each entity type is associated with a set of attributes. For example, the attributes of a process are its creation time, command used

---

[1] https://github.com/syssec-utd/proviot

to invoke, executable path and other relevant information. We use these entities and the interactions between them (*e.g.,* creation, reading, writing) to represent the system behaviors of the IoT device.

The collected data consists of raw syscall sequences which are translated into meaningful system information (*e.g.,* file descriptors are translated into file paths and PIDs are translated into process names) and stored in the provenance database. After translation, the data collection module processes the information into system events, which embodies the interaction between two system entities. Formally, we define a system event as $e_R(n_s, n_d, t)$ where n_s is the source entity, n_d is the destination entity, t is the time when e occurs, and R is the relationship (*e.g.,* read, write, create). For example, `Process A opens (with write permission) File B` at time T is $e_w(A, B, T)$.

System events are queried from an in-memory database to generate ❷ the provenance graphs, $G(p)$, for a particular program. The generated provenance graphs are decomposed into subgraphs (*i.e.,* provenance paths). Formally, we define a causal path λ in a provenance graph $G(p)$ as an ordered sequence of system events (or edges) $\{e_1, e_2, \ldots, e_n\}$ in $G(p)$, where $\forall e_i, e_{i+1} \in \lambda$, $e_i.dst == e_{i+1}.src$ and $e_i.time < e_{i+1}.time$. The time constraint enforces that an event can only be dependent on events in the past, which prevents infinite loops.

After causal paths are extracted from provenance graphs, the relevant causal paths are extracted ❸ using a frequency database. Relevant causal paths during training are the common causal paths since we want to train the behavioral model with common provenance paths, but during anomaly detection relevant causal paths are the rare, since we want to detect these rare behaviors.

The frequency database stores historical behavior information for a particular program and is used during the ranking process, including how many times the system has seen a particular system event in the past. For example, if an entry in the frequency database is <`/bin/bash|CREATE|/bin/cat`, [1000]>, it means in the past `/bin/bash` created `/bin/cat` one thousand times. False positives due to benign program evolution is an important issue for ML-based detectors. Therefore, ProvIoT updates the frequency database at run-time using benign behavior to capture the evolution of program behavior.

The relevant causal paths are converted ❹ to feature vectors using *doc2vec* (Le and Mikolov, 2014). The local model is then trained ❺ on the feature vectors, and the model weights are sent ❻ to the Cloud Brain to update the global model and propagate the localized information to the other connected Local Brain instances. After the Local Brain receives the aggregated global model weights, it starts the anomaly identification process. The Local Brain model uses the new model weights to detect anomalous behavior and raises an alert if any anomalous events are found. The pipeline is visualized in Figure 3.2 and explained in §3.4.

Since the only connection with the Cloud Brain host is for sending and receiving model weights, the network overhead is constant and independent of the amount of data processed on each IoT device. Additionally, since the global models are stored on the device itself, the Local Brain can still operate even if the network connectivity is lost. This gives ProvIoT an advantage over other IoT behavioral anomaly detectors(Cosson et al., 2021; Rieger et al., 2023) as it does not require the transmission of the data to a centralized server for detection to occur. This also preserves the privacy of the device. We describe the detection models in more detail in §3.4.

3.3.2 Federated Architecture: Cloud Brain

Since the Cloud Brain resides in the cloud, it has sufficient computing power to aggregate ❻ the model updates from multiple Local Brain instances to build the global detection models and to synchronize the aggregated global weights with the Local Brain instances. This architecture scales more efficiently than centralized off-device detection schemes because federated averaging is infrequent and is less intensive than performing anomaly detection

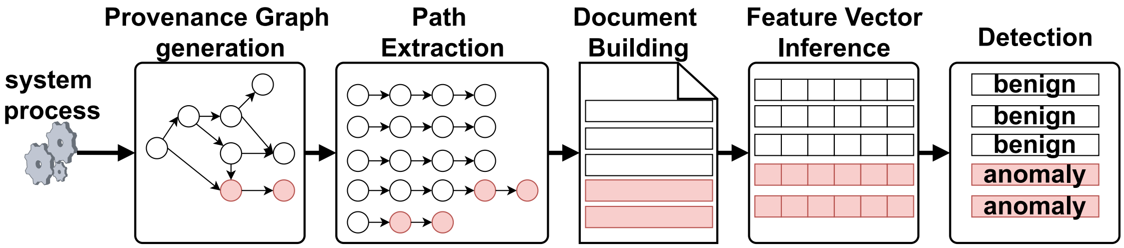


Figure 3.2: The detection pipeline of the Local Brain.

for an entire fleet of IoT devices, so expanding the fleet does not dramatically increase the computational requirements of the Cloud Brain.

Federated Aggregation. Device specific anomaly detection models are aggregates them using the `FederatedAveraging` algorithm described in (McMahan et al., 2017). Because each device gathers data only from the information it encounters, the data from a single device represents a slice of all the potential benign behaviors. The aggregation that takes place in the Cloud Brain improves the detection accuracy by combining the different pieces of information from all the connected clients. Using local and global models and sharing only the model weights solves the problem of maintaining privacy in each device.

3.4 Federated Detection in IoT Domain

A core component of ProvIoT is its ability to perform detection autonomously on the IoT device without a centralized server. The local detection module raises alerts when suspicious events occur. While a centralized server is used to keep the detection module up to date, it is not necessary for detection. The detection pipeline in the Local Brain use the same data collection and preprocessing steps as the training pipeline, but selects rare paths for detection instead of common paths for training.

The detection pipeline, shown in Figure 3.2, works in the following manner: first, the Local Brain will generate provenance graphs for each target program and extract rare causal paths for consideration. These causal paths are converted into causal sentences (Wang et al., 2020), which are combined to form a causal document. Next, we use an NLP model, *doc2vec* (Le and Mikolov, 2014), to embed the causal document as set of k-dimensional feature vectors. Finally, we use the trained *AutoEncoder* (Google, 2021b) model to detect the malicious causal paths as done by recent studies (Han et al., 2021; Mukherjee et al., 2023). The intuition is that when feature vectors are inferred using the *doc2vec* model, benign causal paths will generate feature vectors that would be clustered separately from anomalous feature vectors.

It is possible that there is no anomaly in a process, but a combination of processes can lead to the anomaly, even still ProvIoT would be able to identify these anomalies. Since, during the graph building phase we capture both the forward dependencies (*e.g.,* creating new interactions with different system artifacts or modifying system artifacts) and backward dependencies (*e.g.,* capturing the malware payload deployment event that started the attack as well as different program and data dependency), we obtain a holistic system snapshot. Because malicious activities contain previously unseen behavior, their corresponding causal paragraphs will contain rare sentences, which will be inspected during the detection process.

3.4.1 Provenance Graph Building and Subgraph (Path) Selection

For each target program, the Local Brain will generate provenance graphs from system events gathered in the data collection module. Causal paths are extracted from the provenance graphs through a series of random walks. We consider the rarest 15 % of the causal paths using (Hassan et al., 2019); 15 % was empirically determined in our training phase. Following (Hassan et al., 2019; Wang et al., 2020; Mukherjee et al., 2023), the rarity of a causal path is calculated using the frequency database introduced in §3.3.1. The regularity of an event is $R(e = (u, v, r)) = \frac{|Freq(u,v,r)|}{|Freq(u,*,r)|}$, and the regularity of a causal path is $R(P = (e_1, e_2, \ldots, e_n)) = \Pi_{e \in P} R(e) \cdot \alpha$, where α is a correction factor to prevent the regularity of long paths from trending towards zero. The rarity of a path is simply the complement of its regularity, $1 - R(P)$.

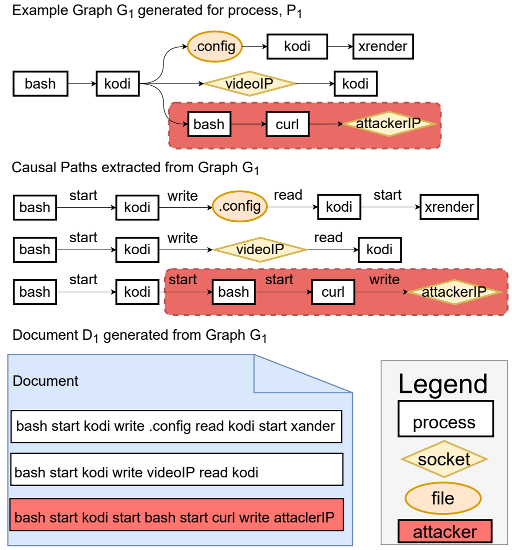

Example Graph G1 generated for process, P1
bash
kodi
.config
kodi
xrender
videoIP
kodi
bash
curl
attackerIP
Causal Paths extracted from Graph G1
bash
start
kodi
write
.config
read
kodi
start
xrender
bash
start
kodi
write
videoIP
read
kodi
bash
start
kodi
start
bash
start
curl
write
attackerIP
Document D1 generated from Graph G1
Document
bash start kodi write .config read kodi start xander
bash start kodi write videoIP read kodi
bash start kodi start bash start curl write attaclerIP
Legend
process
socket
file
attacker

Figure 3.3: Example causal paths extracted from a provenance graph, G_1, generated for process, P_1. Using the extracted causal paths the sentences are formed for a document, D_1.

The information embedded in the provenance graph needs to be extracted to be used as features. One naïve approach may be to use the whole provenance graph for detection. However, using the entire graph will result in a lot of benign noise (events) being mixed into the overall data and the overhead needed to digest the entire graph for ML purposes are unreasonable in an IoT context. Many stealthy malware writers use this property to attempt to blend in with the surrounding benign noise in the graph. Thus, we use a frequency database, as defined in (Hassan et al., 2019) to extract *rare* causal paths from the whole provenance graph. An example of causal paths extracted from a provenance graph in Figure 3.3.

For each selected path, ProvIoT removes the host/entity-specific features, such as host name and identifier, from each node and edge. This process ensures that the extracted representation is general for the subsequent learning tasks.

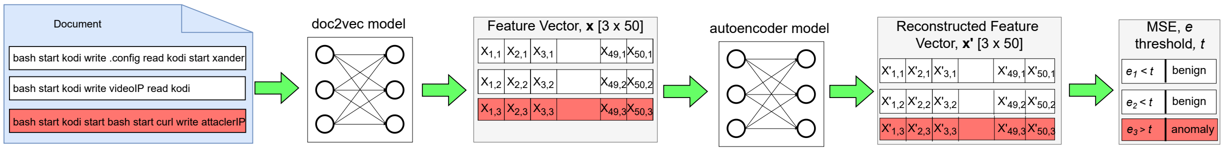

Document
bash start kodi write .config read kodi start xander
bash start kodi write videoIP read kodi
bash start kodi start bash start curl write attaclerIP
doc2vec model
Feature Vector, **x** [3 x 50]
$X_{1,1}$ $X_{2,1}$ $X_{3,1}$ $X_{49,1}$ $X_{50,1}$
$X_{1,2}$ $X_{2,2}$ $X_{3,2}$ $X_{49,2}$ $X_{50,2}$
$X_{1,3}$ $X_{2,3}$ $X_{3,3}$ $X_{49,3}$ $X_{50,3}$
autoencoder model
Reconstructed Feature Vector, **x'** [3 x 50]
$X'_{1,1}$ $X'_{2,1}$ $X'_{3,1}$ $X'_{49,1}$ $X'_{50,1}$
$X'_{1,2}$ $X'_{2,2}$ $X'_{3,2}$ $X'_{49,2}$ $X'_{50,2}$
$X'_{1,3}$ $X'_{2,3}$ $X'_{3,3}$ $X'_{49,3}$ $X'_{50,3}$
MSE, e threshold, t
$e_1 < t$ benign
$e_2 < t$ benign
$e_3 > t$ anomaly

Figure 3.4: Example detection workload for graph G_1 in Figure 3.3. After the document D_1 is formed, the causal sentences in the document are converted into feature vectors (fv) using doc2vec model. Then the fv are fed into the AutoEncoder to get the reconstructed fv. Sentences are flagged as anomalous if the mean squared error between the original fv and the reconstructed fv is above a threshold determined during training.

3.4.2 Document-to-Neural Embedding Model

The extracted causal paths need to be vectorized before they can be processed by the local detection model. As illustrated in Figure 3.3, we first translate the causal paths into causal sentences, a process detailed in (Wang et al., 2020). These causal sentences collectively form a document. Following recent methodologies (Wang et al., 2020; Mukherjee et al., 2023), we employ the *doc2vec* Natural Language Processing (NLP) model (Le and Mikolov, 2014) to transform these causal sentences into their corresponding feature vectors, as depicted in Figure 3.4. Our *doc2vec* model, trained using data from benign deployments, ensures that causal sentences common in benign contexts yield neural embeddings that are more similar to each other compared to embeddings from rare causal sentences.

3.4.3 Federated PIDS: AutoEncoder

In ProvIoT, each Local Brain trains AutoEncoder models on the feature vectors from 3.4.2 and shares the model weights with the Cloud Brain for aggregation using federated averaging (McMahan et al., 2017). After fetching the global AutoEncoder models from the Cloud Brain, the Local Brain is ready to independently detect anomalies.

The Cloud Brain is distinct from the central server in the current state-of-the-art (SOTA) provenance system for IoT (Cosson et al., 2021; Rieger et al., 2023), which collects all the device data over the network and performs anomaly detection serverside. ProvIoT's on-device detection approach affords several advantages: *(1)* sending only the model weights over the network both reduces network overhead and preserves the privacy of activities on the IoT device; *(2)* on-device detection allows the IoT device to remain protected even when disconnected from the network; and *(3)* distributing the detection workload to the edge devices allows ProvIoT to scale horizontally with the size of the IoT device fleet, rather than requiring a vertically scaling central server.

The Local Brain's AutoEncoder models follow a typical structure for anomaly detection. The AutoEncoder has an encoder, which maps the benign feature vectors to a latent space representation that captures behavioral patterns, and a decoder, which reconstructs the original input. To detect anomalies, we measure the Mean Squared Error (MSE) of the reconstructed input and the original input; the input is flagged as anomalous if the MSE is higher than an experimentally determined threshold, which for our implementation was the 99th percentile. The intuition behind this detection scheme is that the AutoEncoder can effectively reconstruct benign samples similar to the ones it was trained on, but should struggle to reconstruct samples that are substantially different (*i.e.,* anomalies).

3.5 Evaluation

In this section, we evaluate ProvIoT's efficacy in detecting stealthy attacks in IoT devices. To this end, we seek answers for the following three research questions (RQs):

RQ1: Detection Accuracy. How effective is ProvIoT at detecting stealthy attacks (*e.g.,* Fileless IoT malware impersonating trusted system programs) and APT campaigns? (§3.5.3, §3.5.4)

RQ2: Benefit of Federated Architecture. What benefits does the collaborative architecture have over a centralized approach? (§3.5.5)

Table 3.1: The IoT applications chosen for evaluation as well as their usage examples.

Usages	Application	Scenario
Voice Assistant	`google`	Inquired general knowledge and everyday household questions to Google Assistant.
Media Center	`kodi`	Updated media streams and played media during different parts of the day.
IP Camera	`motion`	Started streaming multiple live camera streaming server and watched them.
Network Attached Storage	`samba`	Performed network storage action such as list all the files, delete a file, or add a file.
Network Security Monitor	`zeek`	Investigated the network traffic coming from IoT using Zeek.

RQ3: Resource Efficiency. What CPU and memory overhead does ProvIoT incur? (§3.5.6)

3.5.1 Dataset

In this section, we introduce the provenance datasets that consist of provenance graphs generated by capturing the benign and malicious IoT system's behavior.

IoT Workload. The Table 3.1 shows the typical usage for the IoT applications. Typical usage for media center (*e.g.,* `kodi` (kodi, 2018)) is to browse different streams to find playable and downloadable content. `kodi` was used to download different medias from the wed along with browsing different steams. A voice assistant such as `Google Assistant` (Google, 2018) was used for answering common questions such as "what is the weather like?". An IP camera (*e.g.,* `motion` (motion, 2018)) was used to stream our lab setting from our home. We used a network attached storage (*e.g.,* NAS) to access files from remote locations as well as to modify the files. Finally, we used a network security monitoring tool (*e.g.,* `zeek` (zeek, 2021)) to sniff and inspect at the network traffics that was generated in our lab environment.

Dataset Statistics. This section contains the data set details shown in Table 3.2 and Table 3.3. In Table 3.2 the benign dataset is represented where we experimented with five commonly used IoT programs (Costin and Zaddach, 2019) and twenty prevalent Linux

Table 3.2: Number of vertices and edges used to create a benign profile for IoT applications and system programs

	Avg. # of causal paths	Avg. # of total vertices / edges	Avg. # of forward vertices / edges	Avg. # of backward vertices / edges
IoT Application				
`google`	571,052.33	159.0 / 314.0	95.67 / 216.0	63.33 / 98.0
`kodi`	29,946.89	210.33 / 273.78	149.33 / 176.89	61.0 / 96.89
`motion`	9,113.0	179.0 / 504.0	5.0 / 4.0	174.0 / 500.0
`samba`	85,347.0	2,537.0 / 2,857.0	76.4 / 120.8	2,460.6 / 736.2
`zeek`	494,160.0	2,149.5 / 3,724.5	1,032.5 / 1,124.5	1,117.0 / 2,600.0
average	**237,923.84**	**1,046.97 / 1,534.66**	**271.78 / 328.44**	**775.19 / 1,206.22**
System Program				
`bash`	166,355.43	454.25 / 510.76	10.57 / 9.31	443.68 / 501.45
`cat`	184,346.43	310.51 / 210.9	9.0 / 6.99	301.51 / 203.91
`cp`	175,636.86	193.42 / 212.7	179.09 / 184.69	14.33 / 28.01
`cron`	214,827.71	327.16 / 241.85	10.27 / 9.96	316.89 / 231.89
`dash`	153,808.57	371.87 / 381.97	211.61 / 206.44	160.26 / 175.53
`dbus-daemon`	156,713.0	20.16 / 20.04	9.02 / 6.42	11.14 / 13.62
`dd`	213,601.29	995.5 / 1,003.6	551.68 / 501.81	443.82 / 501.79
`firefox`	176,843.86	194.22 / 504.56	15.84 / 18.78	178.38 / 485.78
`grep`	212,413.86	191.51 / 502.32	13.51 / 16.43	178.0 / 485.89
`java`	169,180.71	133.94 / 222.4	17.44 / 19.63	116.5 / 202.77
`ls`	179,185.86	213.62 / 356.47	10.25 / 9.3	203.37 / 347.17
`nginx`	258,367.17	514.27 / 514.13	500.76 / 501.26	13.51 / 12.87
`perl`	809.0	25.01 / 23.22	11.95 / 12.05	13.06 / 11.17
`ps`	181,846.43	834.01 / 998.14	369.21 / 501.77	464.8 / 496.37
`python`	161,755.57	365.71 / 348.31	11.51 / 8.14	354.2 / 340.17
`rm`	174,590.43	452.89 / 440.38	15.06 / 18.5	437.83 / 421.88
`service`	231.43	18.32 / 21.24	15.32 / 18.55	3.0 / 2.69
`sh`	208,367.43	445.01 / 851.27	4.16 / 357.78	440.85 / 493.49
`smbd`	201,559.57	355.37 / 371.15	9.69 / 3.39	345.68 / 367.76
`sshd`	182,601.57	233.04 / 234.15	9.35 / 6.6	223.69 / 227.55
average	**168,652.11**	**332.49 / 398.48**	**99.26 / 120.89**	**233.23 / 277.59**

Table 3.3: Number of vertices and edges used to create IoT Malware and APT attack profile

	Avg. # of causal paths	Avg. # of total vertices / edges	Avg. # of forward vertices / edges	Avg. # of backward vertices / edges
IoT Malware				
`BASHLITE`	110.5	21.0 / 21.0	4.0 / 3.0	17.0 / 18.0
`FritzFrog`	46,253.8	751.0 / 747.4	248.6 / 246.8	502.4 / 500.6
`lizkebab`	293.2	29.0 / 33.0	6.0 / 4.0	23.0 / 29.0
`randomware`	250.4	28.0 / 44.0	8.0 / 12.0	20.0 / 32.0
average	**11,726.98**	**207.25 / 211.35**	**66.65 / 66.45**	**140.6 / 144.9**
APT Kill Chain Scenario				
Gain Access (S1)	2,789.6	510.6 / 554.8	495.2 / 537.6	15.4 / 17.2
Establish a Foothold (S2)	46,763.75	470.25 / 550.12	398.38 / 429.5	71.88 / 120.62
Deepen Access (S3)	1,192.4	171.0 / 202.6	164.0 / 195.0	7.0 / 7.6
Move Laterally (S4)	27,314.33	97.5 / 116.0	70.17 / 84.83	27.33 / 31.17
Look, Learn and Remain (S5)	20,521.75	928.12 / 983.5	897.38 / 929.62	30.75 / 53.88
average	**19,716.37**	**435.49 / 481.40**	**405.03 / 435.31**	**30.47 / 46.09**

Table 3.4: APT TTPs for cyber-killchain stages

Cyber-killchain Stages	Techniques (ATTCK TTP)	Scenarios
Gain Access (S1)	Exploitation for Client Execution (T1203)	Attackers modify a benign looking executable, but once the user opens the application it can be used by the attacker for arbitrary code execution
	File and Directory Permissions Modification (T1222)	Attacker modifies objects in the system so that it can be copied by lower privilege users that the attacker has hijacked
Establish a Foothold (S2)	Data from Local System (T1005)	Attacker moves around the file system, finding files that contain valuable information
	Exfiltration Over C2 Channel (T1041)	Attacker downloads valuable files into a local directory
Deepen Access (S3)	Create and Modify system process (T1543)	Attacker creates a system process that can run in the background and do reconnaissance or mine information
	Service Stop (T1489)	Attacker stops firewall or external IDS services so that they cannot detect the APT
Move Laterally (S4)	Process injection (T1055)	Attacker injects a vulnerable process such as a trojan into a benign application so that IDS cannot differentiate
Look, Learn, and Remain (S5)	System Information Discovery (T1082)	Attacker discovers system hardware information so that they can craft better exploits or exploit hardware vulnerabilities
	Network Service Scanning (T1046)	Attackers scan network services to find services they can use as backup or use as a secondary mode of connections
	Network Sniffing (T1040)	Attackers sniff the network to find insecure SSL connections or any other connections to extract valuable information

system programs (Liu, Zhang, Li, Jee, Li, Wu, Rhee, and Mittal, Liu et al.). Table 3.3 shows the malicious data set which consists of two parts: four IoT malware which impersonated the twenty Linux system programs and APT kill chain scenarios conducted using the five IoT programs.

APT Scenarios. Advanced Persistent Threat (APT) scenario was established in our malicious testbed by loading APT kill-chain components using Fileless wrapper (Table 3.4). The APT attack vectors were coordinated to comprise the end-to-end attack campaign referring to MITRE ATT&CK framework.

Dataset Components. Our datasets consist of three major components: forward graphs, backward graphs and causal paths. The forward graphs consist of all the system events that are caused by the process associated with a Point of Interest (POI) event, *e.g.,* process creation, file and socket reads/writes. The backward graphs consist of the system events that created the POI event. We merge the forward and the backward graphs to get a unified graph that captures all the system events associated with the POI event. We then extract causal paths from this unified graph; the size statistics for the graphs and causal paths are shown in Table 3.2. To generate a graph dataset for a given program, we use all process creations for the given program name as POI events to build forward and backward graphs.

Benign Dataset. We consulted our university's Institutional Review Board (IRB) to develop an ethical experimental protocol for selecting volunteers for benign data collection. Once the volunteers were chosen, they received information about how their data would be used and securely stored to ensure confidentiality. The benign data collection took place over a period of twelve months, from January 2021 to December 2021, and resulted in the collection of over 30 TB of data. The benign profile for the programs was constructed by gathering system events from a diverse set of 33 devices, including ARM-based IoT devices such as Raspberry Pi, Google TPU, and NVIDIA Jetson Nano boards (rpi, 2018; Google, 2021a; NVIDIA, 2022). The device platforms consist of 1 Google TPU, 1 NVIDIA Jetson Nano, 3 Raspberry Pi 4, 5 Raspberry Pi 3B+, 5 desktops, 5 laptops, and 13 servers. Importantly, the provenance graphs that capture the behavior of a given system program exhibit a relative consistency across different IoT devices and platforms.

The IoT devices in our benign testbeds performed various IoT tasks and common system operations categorized as *IoT Applications* and *System Programs* respectively in Table 3.2. Using this system event data, we generated provenance graphs for popular IoT applications (Bansal et al., 2022) and common system programs (Cozzi et al., 2018a; Costin and Zaddach, 2019) that are frequently targeted for impersonation. We chose 1000 benign process instances for each of the 20 programs and 150 instances for each of the 5 IoT applications to create the benign dataset. The provenance graphs generated from the benign IoT applications consisted of 237,923.84 causal paths, 1,046.97 vertices, and 1,534.66 edges (IoT Application in Table 3.2) on average. Similarly, the provenance graph generated from the Linux system processes had an average of 168,652.11 causal paths, 332.49 vertices, and

398.48 edges (System Program in Table 3.2). For readers interested in further details about the statistics of the benign dataset and how it was generated.

Malicious Dataset. We created two isolated testbeds to run the malicious workloads. Firstly, we launched publicly known IoT malware using a Fileless wrapper (Cybersecurity, 2021) to impersonate the identities of the popular IoT applications in Table 3.1. Second, we conducted a typical APT scenario by carefully coordinated the APT attack vector with the MITRE ATT&CK (ATT&CK®, 2022b) framework to comprise the end-to-end attack (ATT&CK®, 2022b) campaign. We launched a stealthy attack campaign that contains five kill-chain(Cyb, 2021) stages (Table 3.4) — *(S1) gain access* by injecting a malicious payload into an active benign process; *(S2) establish a foothold* by communicating back to a C&C server over HTTPS (port 443); *(S3) deepen access* using a privilege escalation exploit (Metasploit, 2021), *(S4) move laterally* by scanning the local network for vulnerable hosts with open ports; and *(S5) look, learn, and remain* by exfiltrating sensitive user data to the C&C server. We refer to the Cyber-killchain (Cyb, 2021) framework to include essential components that comprise the successful multi-stage APT campaign. Each attack stage was conducted by different attack TTPs using Metasploit (Metasploit, 2021).

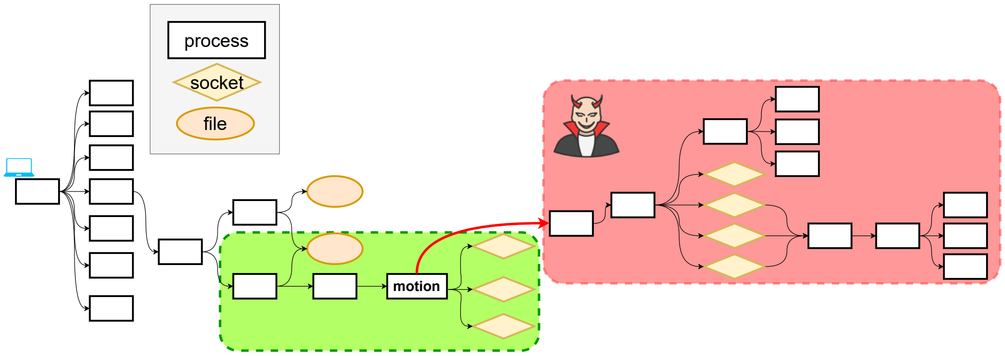


Figure 3.5: Attacker injects and creates Fileless malware as a child process of `motion` process. The provenance graph captures the attacker's behavior which can be used for detection.

We injected each attack TTP into five common IoT applications listed in Table 3.1 using a Fileless wrapper (Cybersecurity, 2021). Therefore, the IoT application's behavior captured in the provenance graph would contain additional nodes and edges (*i.e.,* malicious subgraphs) corresponding to the malicious behavior due to the injected attack TTPs. Because the malicious payload behaves differently than the benign application behavior, those malicious subgraphs are likely to contains rare and anomalous paths that will be detected by the Local Brain. In Figure 3.5, we render the simplified provenance graph where we injected one of the attack TTPs to `motion`. It adds a subgraph whose size is proportional to the number of malicious activities performed.

We performed the program impersonation experiment five times for each of the four Fileless IoT malware samples, with a total of twenty impersonation targets (Table 3.5), resulting in a total of 400 experiments. We conducted the APT scenario seven times on each of the five APT attack stages for five IoT applications (Table 3.4), totaling 175 experiments to build the APT dataset. Combining all our experiments, we conducted a total of 575 experiments (175 APT + 400 malware) to create the anomalous dataset. The provenance graphs collected from the malware evaluation have an average of 11,726.98 causal paths, 207.25 vertices, and 211.35 edges. The provenance graphs for the APT Kill chain scenario have an average of 19,716.37 causal paths, 435.49 vertices, and 481.40 edges.

We obtain (2) FritzFrog samples, a real-world Fileless IoT malware we discussed in Section 2.8 to evaluate ProvIoT against their impersonation targets of `Nginx` server and `ifconfig`. The real-world malware experiment using FritzFrog was conducted 10 times for each impersonation target (*e.g.,* `Nginx` server and `ifconfig` program).

3.5.2 Experimental Protocol

To generate the training and validation sets, we extract all the causal paths from the provenance graphs generated during benign deployment, reserving 90% of the data for training and 10% for validation. To generate the test set, we extract the rarest 15% of causal paths from the malicious testbeds, which simulates a real environment that has been attacked (Hassan et al., 2019; Wang et al., 2020) and includes a mix of benign and anomalous paths. The Local Brain instances train on the benign training data and propagate their model weights to the Cloud Brain. The Cloud Brain then performs federated aggregation on those models to generate a global model, then propagates the global model back to the Local Brain instances. Each Local Brain tunes its detection threshold using its own validation set. In intrusion detection, we emphasize the importance of *unsupervised* learning because the defender should not make strong a priori assumptions about the attacker's behaviors.

After we generate the provenance graphs for the benign and malicious cases, we extract all the causal paths for the benign programs and choose only the top 15% of the rarest paths (empirically determined) for the malicious cases like previous studies (Hassan et al., 2019; Wang et al., 2020). The extracted paths are then converted into feature vectors and split into eighty-ten-ten ratios of training, testing, and validation sets. The Local Brain first trains on the benign feature vectors and sends the local models' weights to the Cloud Brain for aggregation. The Cloud Brain updates the global models and propagates the global models' weights back for detection. The threshold used for detection is hypertuned using the benign validation dataset. We believe our *unsupervised* modeling is a reasonable design choice for security applications where we cannot make any concrete assumptions on the attack behaviors, and program behavior can change rapidly.

3.5.3 Fileless Malware Detection

To represent a wide variety of malware, we selected two popular IoT malwares from (Ding et al., 2020), a natively Fileless IoT malware (Harpaz, 2020), and a typical ransomware that would target an IoT system. We injected these well-known IoT malwares into trusted system processes using a Fileless wrapper (Cybersecurity, 2021) to impersonate them. The detection results, summarized in Table 3.5, demonstrate that ProvIoT achieves high F1 scores for the majority of combinations, ranging from 0.96 to 0.98. This indicates that

Table 3.5: ProvIoT is highly effective in distinguishing IoT malware impersonating as benign system process as evident from high F1 scores. Grey cells contain low F1 score to indicate indistinguishable malware behavior for system process, discussed in §3.5.3.

Impersonation target	Malware			
	`BASHLITE`	`FritzFrog`	`ransomware`	`lizkabab`
`bash`	0.98	0.96	0.96	0.98
`cat`	0.93	0.99	1.00	0.97
`cp`	0.92	0.97	0.92	0.95
`cron`	0.97	0.98	0.98	0.97
`dash`	0.95	0.96	1.00	0.98
`dbus-daemon`	0.94	0.95	0.92	0.98
`dd`	0.96	0.97	0.98	0.99
`firefox`	0.97	0.96	0.99	1.00
`grep`	0.96	0.97	0.94	0.95
`java`	0.96	0.96	0.96	0.98
`ls`	0.99	0.96	0.94	0.98
`nginx`	0.97	0.98	0.98	0.96
`perl`	0.96	0.96	0.95	0.97
`ps`	0.98	0.97	0.95	0.97
`python`	0.93	0.97	0.93	0.99
`rm`	0.92	0.96	0.93	0.98
`service`	0.93	0.95	0.90	0.99
`sh`	0.96	0.97	0.91	0.98
`smbd`	0.96	0.96	0.99	0.99
`sshd`	0.97	0.96	0.97	0.98
Average	**0.96**	**0.97**	**0.96**	**0.98**

even when IoT malware is Fileless and impersonates benign programs, its behavior remains distinct from the original system behavior.

However, some (impersonation target, malware) pairs, highlighted in Table 3.5, proved challenging for ProvIoT to reliably detect: `BASHLITE` was able to effectively masquerade as `cp` and `rm` because it primarily performs file copy and delete operations on the local device while preparing to participate in the botnet; `ransomware` effectively impersonated `cp` with large amounts of file copy operations, `dbus-daemon` with significant inter-process communication for cryptographic exchanges, `service` with manipulation of antivirus services, and `sh` with command execution.

To evaluate ProvIoT's detection against FritzFrog, we ran our attack and evaluated the result, producing both `Nginx` and `ifconfig` cases. As shown in Table 3.6, ProvIoT's

Table 3.6: Federated attack scenarios result for FritzFrog attack

Malware	Target	Precision	Recall	F1
`FritzFrog`	`nginx`	0.93	1.00	0.99
	`ifconfig`	0.97	1.00	0.99
Average		**0.95**	**1.00**	**0.99**

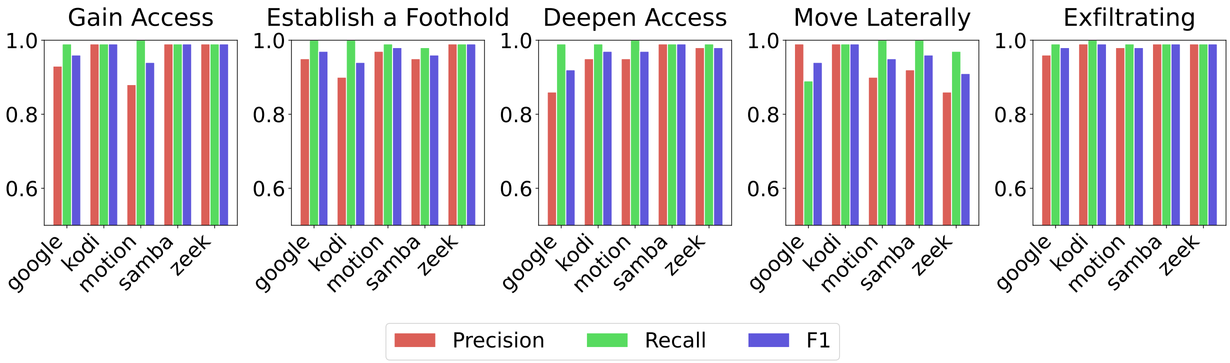


Figure 3.6: High detection accuracy of ProvIoT against APT attacks using federated learning, some rare exceptions which are discussed in §3.5.4.

precision ranges from 0.93 to 0.97, recall is 1, and F1-score is 0.99. This shows ProvIoT is efficient in detecting real-life Fileless malware.

3.5.4 APT Campaign Detection

The consistently high detection accuracy (Wang et al., 2020; Han et al., 2021) of ProvIoT, as measured by precision, recall, and F1 score, is showcased in Figure 3.6. Outside some rare exceptions, which will be discussed in more depth, the precision ranges from 0.93 to 0.99, the recall ranges from 0.97 to 1, and F1 scores range from 0.95 to 0.99. The results show that ProvIoT can reliably detect APT attacks while limiting the number of false alarms.

ProvIoT generates more false positives than false negatives, evidenced by its higher average recall (99%) than average precision (95%). This trend is also seen in other anomaly detection systems(Wang et al., 2020; Han et al., 2021). The high F1 score shows that the threshold is chosen in such a way that the actual anomalous behaviors (true positives) are detected rather than reducing FPs. Therefore, ProvIoT does not compromise on its detection ability to address false positive rates.

False Negative Cases. Even with path-based behavioral modeling, certain attack cases (*e.g., move laterally (S4)* attack for `google`) are hard to detect because the attacker's behavior is extremely similar to the application's benign behavior. The precision is 0.99 and the recall is 0.89, which is much lower than the second-lowest recall rate of 0.97. The *move laterally (S4)* stage scans for vulnerable ports to exploit, which is behaviorally similar to `google` scanning ports for available IP cameras.

False Positive Cases. ProvIoT has delivered steady and robust detection performance across our various APT workloads (Table 3.1). Against some APT stages, ProvIoT had a relatively high false positive rate such as *Deepen Access (S3)* for `google` has precision of 0.86 and recall of 0.99, *Establishing a Foothold (S2)* for `kodi` has precision of 0.90 and recall of 1, *Gain Access (S1)* for `motion` has precision of 0.88 and recall of 1; *Move Laterally(S4)* for `samba` has precision of 0.86 and recall of 0.99 and `zeek` has precision of 0.86 and recall of 0.97.

These instances of high false positive rates are due to system interactions with high behavioral variance. We investigated these cases and outlined the explanations based on the ground truth:`kodi` often reads hidden configuration files, downloads files containing streaming links from the internet and writes them to temporary locations; `google` creates and stops many short-lived threads; `motion` changes directory and file permission configuration for camera video storage; `samba` and `zeek` both scan and listen to different IPs and ports, which generates noisy provenance graphs (high variance). We see a high rate of false positives surrounding the creation and modification of temporary files and directories; since these behaviors are rare and not well-represented in the benign dataset, so they are marked as anomalous even when the actions are not malicious. The majority of the malicious paths were marked correctly as anomalous even though the precision score was below 0.90, the recall score was above 0.96. These results show that ProvIoT is very effective in detecting stealthy malware.

3.5.5 Federated Learning Benefits

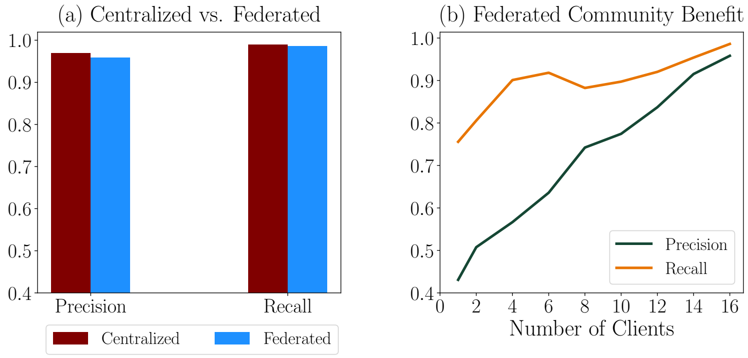


Figure 3.7: (a) Federated performance is similar to centralized performance on the same data. (b) Increasing the number of clients increases performance by increasing the amount of data in the system.

We evaluate ProvIoT's federated approach against a traditional centralized architecture using `kodi` as a representative application. Figure 3.7(a) shows that ProvIoT trades just 1% precision for the scalability, privacy, and reliability benefits of the federated architecture. The centralized model was trained on the full dataset and achieved 0.97 precision and 0.99 recall. For ProvIoT, we used the 16 clients from our benign deployment that had `kodi` installed for training, then evaluated those models in our malicious testbeds. In this experiment, ProvIoT achieved 0.96 precision and 0.99 recall, performing almost identically to the centralized approach.

To demonstrate how ProvIoT is able to overcome the data view limitation of provenance-based anomaly detection on IoT devices, we visualize the average performance of the Local Brains as more clients are incorporated into the system in Figure 3.7(b). By adding new Local Brains that see different data, the Cloud Brain is able to aggregate the incoming models to export a global model that better understands the full benign distribution. These model improvements manifest in improved recall and precision as new clients are introduced.

ProvIoT's federated approach provides critical benefits for IoT in data localization and privacy. The primary security benefit is localized detection, which reduces network overhead, allows detection in the absence of a network connection, and distributes the global detection workload across the federated devices. Further, because we only share model weights, specific system events are not shared with the network, which preserves the privacy of the data.

3.5.6 ProvIoT Overhead

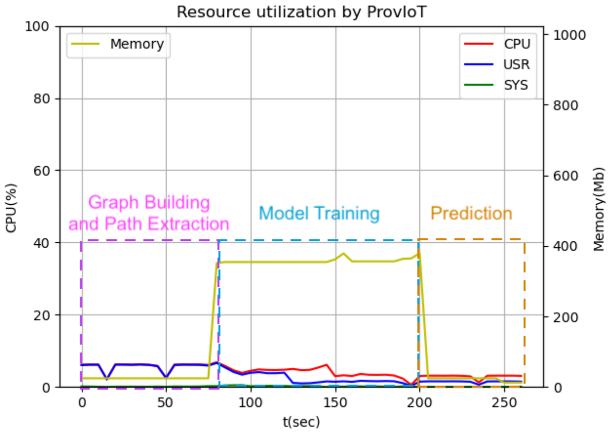


Figure 3.8: On RaspberryPi 4B, Local Brain's processing and prediction uses <10% CPU and 65MB memory. Model training takes about 375MB memory and <10% CPU.

We experimentally demonstrate the overhead imposed by ProvIoT using an event database containing 7,085 process creation events, 56,587 file interactions, and 3,608 network interactions. This is typical for 24 hours of execution. We experimented using different ARM IoT devices such as RaspberryPi 4B board (rpi, 2018) with four CPU cores and 8 GB memory for CPU only device; Jetson Nano (NVIDIA, 2022) with four CPU cores, 4 GB memory and NVIDIA gpu; and Google Edge TPU (Google, 2021a) with single core, 512 MB memory, and edge TPU ML accelerator. To train a reliable model for `kodi`, ProvIoT requires two weeks worth of data, which results in 5.46 GB of data and four weight synchronizations.

To accurately characterize the overhead imposed on the edge IoT devices, we need to consider two different modes of execution: training and detection. Training occurs infrequently (approximately once per week) and requires less than 10% of the CPU processing power and less than 375MB of memory for less than four minutes as shown in Figure 3.8. Detection occurs frequently (approximately once per day) and requires less than 10% of the CPU processing power and less than 65MB of memory for less than two minutes as shown in Figure 3.8. Even during peak resource utilization *i.e.,* during training, PROVIOT does not monopolize the IoT resources. Many home IoT devices, such as smart fridges, thermostats, and doorbells (nes, 2015; app, 2015; sam, 2022; ama, 2020; goo, 2017) contain sufficient memory to support on-device training. Specifically, training and detection have a common provenance graph building and path extraction phase, followed by model training phase or a prediction phase depending on the mode of execution. Even during peak resource utilization *i.e.,* during training, PROVIOT does not monopolize the IoT resources.

3.6 Related Works

IoT Security. With the growth of IoT, a significant number of vulnerabilities have been identified in IoT devices (mirai, 2016; vpnfilter, 2018; Sikder et al., 2021), protocols (zigbeeflaw, 2015), applications, and platforms (Fernandes et al., 2016). In response to IoT attacks, diverse detection and prevention approaches have been proposed, such as network-based solutions (Sivaraman et al., 2015), platform-based solutions (Cosson et al., 2021; Sikder et al., 2017, 2020; Rieger et al., 2023) and application-based solutions (Jia et al., 2017; Wang et al., 2020). Our work defends against stealthy attacks including Fileless malware and APTs targeting IoT devices.

Cosson et al. (Cosson et al., 2021) and Rieger et al. (Rieger et al., 2023) has proposed a centralized node-level monitoring system for IoT using network traffic. However, it requires the local devices to send their local data to a centralized server where the detection occurs. ProvIoT has a major advantage over (Cosson et al., 2021) because the users' data does not leave the local device and detection occurs on the local device without a network connection. (Mothukuri et al., 2021; Shahid et al., 2021; Nguyen et al., 2018) have showed how to do federated anomaly detection on IoT, but solely focused on network data. While the network data is important, stealthy attacks can easily circumvent those defenses with specially crafted network packets. To the best of our knowledge, we are the first to propose a federated, privacy preserving, collaborative learning framework using host-level provenance data for IoT. (Mukherjee et al., 2023) found that the adversarial manipulation of provenance graphs is substantially more challenging than the manipulation of network data. Because provenance graphs effectively capture the runtime behaviors of processes, blending in with benign processes requires substantial domain knowledge, technical expertise, and engineering effort from the attacker.

IoT Defenses. General intrusion detection (Ding et al., 2020; Wang et al., 2020) approaches have been extensively studied. For example, (Bostani and Sheikhan, 2017) and (Raza et al., 2013) designed defenses to detect routing attacks. However, their work focuses on the 6LoWPAN protocol. Our work focuses on creating a generalized federated framework for IoT.

Recently, several anomaly-based solutions have been proposed to detect different IoT attacks. SDN-based approaches (Ozcelik et al., 2017), signature-based approaches (Kumar and Lim, 2019) and machine learning based approaches (Meidan et al., 2018; Bahşi et al., 2018; Nguyen et al., 2018; Nõmm and Bahşi, 2018; Chawathe, 2018) have been proposed to detect IoT botnet attacks such as Mirai. However, these approaches only focus on analyzing network traffic, limiting their capability in detecting attacks with minimal network footprints.

The most directly related previous work is (Cosson et al., 2021; Rieger et al., 2023), which forwards telemetry data for the entire IoT fleet to a central server for anomaly detection; ProvIoT improves upon the privacy and scalability of (Cosson et al., 2021; Rieger et al., 2023) by enabling on-device detection with federated learning and provenance analysis.

CHAPTER 4

EVASIVE ATTACK GENERATION FRAMEWORK – PROVNINJA

4.1 Problem Statement

In this research, we aim to answer the following research question: *Can an adversary use publicly available information and domain knowledge to efficiently evade ML-based detectors?* Leveraging statistical properties of program execution profiles, ProvNinja generates evasive modifications to attack chains. We implemented this general technique to create attacks against four popular ML detectors: *(1)* path-based models — ProvDetector and SIGL ; and *(2)* graph-based models —– GAT and Prov-GAT . As mentioned in §2.4, graph-based models accept entire provenance graphs as input, while path-based models use paths extracted from the graphs. These evasive attacks seek to evade detection (*i.e.,* produce false negatives) by mimicking the execution of benign programs. ProvNinja neither poisons nor interferes with model training and does not intend to generate false alarms.

While we primarily consider the black-box model, referring to the publicly available datasets to build a reference surrogate model, we also explore various assumptions about adversarial knowledge. In §4.8, we evaluate white-box and blind attack scenarios as well. Across four different ML models, we tackle the research challenge in three stages: *(1)* identify conspicuous events in the original attack, *(2)* search for and substitute inconspicuous replacement events, and *(3)* realize the evasive attack and launch it against real-world systems. Our research question suggests an optimization problem: find a function that provides graph transformations that minimize the anomaly detection probability. The function takes as input the established model, the desired attack vector, and system event frequency data, then return a modified attack vector that minimizes the anomaly score of the attack, which decreases the risk of detection (Pierazzi et al., 2020).

4.2 Threat Model

Adversary knowledge. ProvNinja assumes a *black-box* model in our implementation. To avoid alerting the victim, the attacker should aim to minimize the number of black-box model queries required to generate evasive attacks. The model prediction results are only used to determine when to finish improving the attack. The adversary has access to publicly available program execution frequency statistics, which we call the surrogate frequency database. We infer the rarity of events directly from the regularity scores calculated using the surrogate frequency database (Hassan et al., 2019; Wang et al., 2020) (§4.4). This approach typically captures a superset of the edges that strongly influence the model's prediction.

Focusing primarily on the black-box model for the adversary's knowledge and capabilities, we recognize its substantial practical value. Many commercially deployed security solutions deliver pre-trained models to end-user devices. For example, EDR for mobile and desktop computers deploys security models to end-user devices, leading us to reasonably assume that determined adversaries would use them as oracles. We also evaluate the blind and white-box models to provide a complete landscape. For the blind attack model, we eliminate the adversaries' ability to query detection models and rely solely on statistical approximation. In cases of white-box attacks, where the adversary has complete access to the detection model, including model architecture and parameters, we employ the GNN explainer (Ying et al., 2019) to expedite the process of identifying conspicuous events.

Adversary capability. Using this knowledge, the attacker is able to evaluate and realize the feature-level attacks suggested by ProvNinja; while these suggestions are likely to evade the detection model, the difficulty of actualizing the suggested attacks with concrete system actions can vary widely (Pierazzi et al., 2020). We assume a highly skilled and motivated adversary who is capable of devising these stealthy evasive attack vectors.

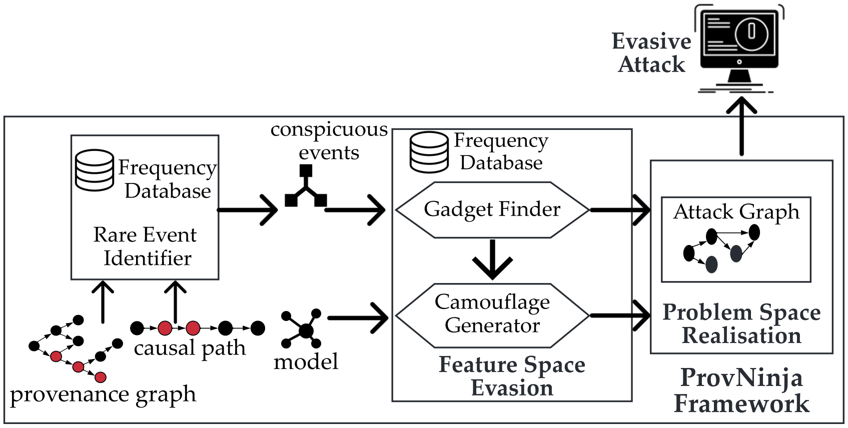


Figure 4.1: ProvNinja framework.

4.3 ProvNinja Overview

There exists a rich literature on adversarial attacks against ML models (Carlini, 2019; Pierazzi et al., 2020) that affect prediction results with minimal overhead. However, previous exploration of adversarial attacks that can exploit provenance-based threat detectors (Goyal et al., 2023) have been hampered by the limited availability of public datasets and the significant effort required to realize such attacks in the problem space.

The core of the evasion mechanism is outlined in Algorithm 1. Compared to other modeling approaches, where the problem space is similar to the feature space, provenance graphs and their feature embeddings are the product of a long series of transformations and summaries of the original problem space system events. To evade provenance-based ML detectors, ProvNinja proposes a three-stage approach as shown in Figure 4.1. First, ProvNinja locates conspicuous edges that can be modified to evade detection. Second, ProvNinja searches for feature space modifications to generate an evasive attack. Finally, we realize the feature space attacks in the problem space to launch the attacks against real systems (Pierazzi et al., 2020).

To the best of our knowledge, ProvNinja is the first to systematically study adversarial evasion of provenance-based ML security detectors using a publicly available surrogate dataset. ProvNinja employs a data-driven approach to construct evasive attack vectors with minimal human oversight while adhering to realistic system constraints. We thoroughly evaluate ProvNinja against various ML models using our comprehensive benign and malicious dataset collected from real-world deployments. To benefit the community and facilitate future research, we will make our dataset publicly available [1] and offer data collection support to researchers and practitioners.

4.4 Program Profile Generation – Frequency History of Events

By removing timestamps and non-essential attributes from the original provenance dataset, we generate a lightweight summary of site-specific event frequencies. Following previous works (Wang et al., 2020; Hassan et al., 2019; Han et al., 2021), this frequency database stores the number of historical occurrences of single-hop relationships between processes, files, and network sockets. For instance, [`/bin/bash/|CREATE|/bin/cat`, 1000] means that `/bin/bash` has created a `/bin/cat` 1000 times in the past. In previous research (Hassan et al., 2019; Wang et al., 2020), the frequency database is used by the defender to calculate the rarity (*i.e.,* potential malice) of system events. This approach complements provenance analysis because each system event is an edge in the provenance graph. One of the most important applications of the frequency database is to provide a program profile that characterizes the site-specific runtime behavior. Referring to the frequency database, we can estimate the typical runtime behavior of a benign instance of a given program. In §4.6.2, we use this information to mimic benign process behaviors.

These use frequency databases to store historical behavior information regarding processes, files, and network connections in a frequency database (Hassan et al., 2019). We assume that system events that have occurred in the past are likely to be the normal behavior of the system. For example, (src, dst, rel). *e.g.,* , [`\bin\bash\CREATE\bin\cat`, 1000]

---

[1] https://github.com/syssec-utd/provninja

means that the process creation path of `\bin\bash` with `\bin\cat` has occurred one thousand times in the past. Comparing against such data, the detection system can then calculate the rarity of a new system event which in turn can indicate potential malice.

We use the frequency database to calculate the *regularity scores* (Hassan et al., 2019; Wang et al., 2020; Han et al., 2021) for a system event, e as well as for the entire causal path $\lambda_{attack} = e_1, e_2, ..., e_l$ of length l. $RS(e) = \frac{Freq(e)}{Freq_{src\_rel}(e)}$, and $RS(\Lambda) = \prod_{i=0}^{l} RS(e_i)$ where $Freq(e)$ is equivalent to how many times the system event e has occurred in the historic window with all 3-tuples (src, dst, rel) of e being the same and $Freq_{src\_rel}(e)$ is equivalent to the frequency of system event e where only src and rel from the 3-tuples are the same. If event e has only occurred rarely in the system, then the value of $RS(e)$ would be close to 0. A low regularity score for a rare system event e_{rare} would propagate a low regularity score for the causal path λ_{attack} which includes the rare event, e_{rare}. An attacker can then utilize this information of how rare the events are in the attack paths to construct a new attack path λ_{gadget} replacing rare system events with common events from the frequency database. In this way, he desired action of the original attack path is preserved making the attack feasible in the real world while simultaneously lowering the detectability of the path.

4.5 Identifying Conspicuous (or Rare) Events

We define "conspicuous" events to be the subset of rare events that also contribute heavily to a given model's prediction. When conspicuous events are replaced with common events, the model's prediction is likely to shift towards benign. Leveraging the surrogate frequency database, we define the regularity of an event as $R_e(u, v, r) = \frac{|Freq(u,v,r)|}{|Freq(u,*,r)|}$ (Hassan et al., 2019). That is, the regularity of an event is the proportional representation of that event among all events with the same source and relation (*e.g.,* of all processes created by `outlook.exe`, what proportion of them were `excel.exe`?). We calculate the regularity score for each event in the attack path and select the k least regular events for replacement. For instance, in our Enterprise APT scenario, some conspicuous events are (`java.exe`, `notepad.exe`, create process) and (`notepad.exe`, `IP:445`, receive from socket).

When trained on just the graph structure, GNN model does not know the node label such as executable name of the process, file names, or socket addresses. But from the structure, the process chains in which the first process reads a file and the last process connects to a socket are distinctive so it has an negative impact on the prediction. Therefore, given this graph, GNN classifies it as anomalous.

identify among the important edges which is rare. For example, from the enterprise APT GNNExplainer identified these process chain as `USERINIT.EXE` → `USERINIT.EXE` → `EXPLORER.EXE` → `OUTLOOK.EXE` → `EXCEL.EXE` → `JAVA.EXE` as important. But, `USERINIT.EXE` → `USERINIT.EXE` → `EXPLORER.EXE` → `OUTLOOK.EXE` events are common and does not need to be replaced. Also, even if the attacker creates an attack that replaces these edges, the attack is infeasible since the attacker cannot modify past causal events with respect to the attacker's entry point. We want to note that from the explanation, *important edges are not always rare*. Therefore, we use the frequency database to find the rare edge, which in this case is `EXCEL.EXE` → `JAVA.EXE`.

The targeted perturbations with the realistic constraints enforced by frequency database create adversarial example that mislead the GNN into misclassifying the pattern as benign. We will discuss more in detail of what we mean by perturbations and realistic constrains in the following sections.

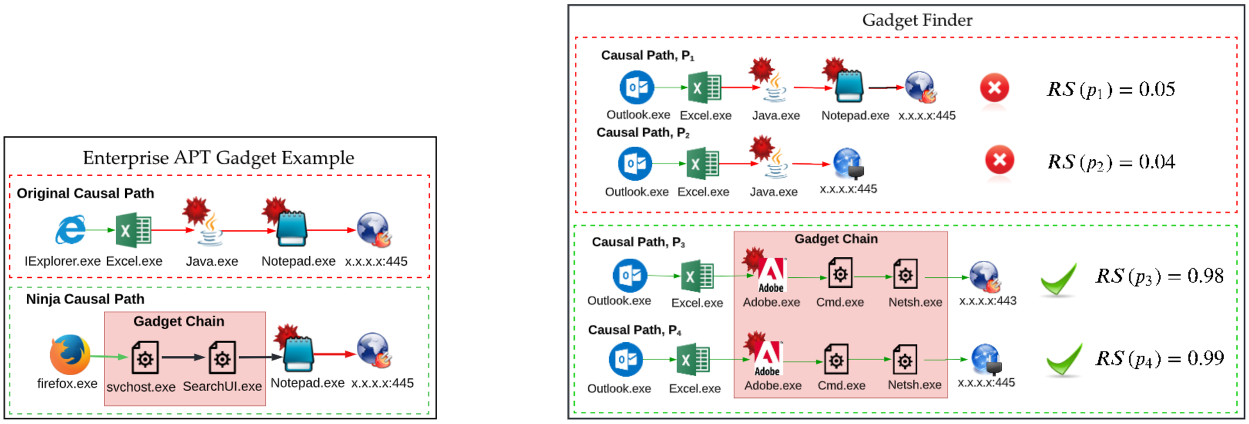
4.6 Feature Space Evasion

When modifying conspicuous events, the goal is to minimize the chances of detection while achieving the same attack objectives. Algorithm 1 shows the general ProvNinja framework that is used to find the evasive adversarial examples. We refer to the surrogate frequency database to find ”gadgets” (further described in §4.6.1) that can replace conspicuous events.

Algorithm 1: ProvNinja

```
Input: Provenance Graph, $G = (V, E)$
Frequency Database, $F$
Defense Model, $M$
Max. Modification Distance, $D$
Regularity Threshold, $T$
Event Search Limit, $K$
Output: Modified Provenance Graph, $G'$, that is classified as benign by $M$, or $\varnothing$ if
        no such graph is found
if $D \leq 0$ then
    return $\varnothing$
$rare\_edges$ =TopRareEdges$(E, K)$
$gadgets = \bigcup_{e \in rare\_edges}$FindGadgetChains$(e, F, T)$
foreach $g \in gadgets$ do
    $G'$ =ApplyGadget$(G, g)$
    if $M(G') == benign$ then
        return $G'$
    $G'$ =ProvNinja$(G', F, M, D - 1, T, K)$
    if $G' \neq \varnothing$ then
        return $G'$
return $\varnothing$
```

An effective gadget achieves the same objectives as the original event, but is more common in benign execution and is therefore less anomalous. The intuition behind gadgets is that an evasive attack should behave as closely to benign activity as possible while still achieving the adversary's objectives. Because path-based ML detectors lose surrounding structural information, gadgets alone are sufficient for evasive attack generation. Graph-based ML detectors, however, will easily detect "naked" gadgets, which typically include sequences of process creations with no intermediate activity. To mimic the structure of benign activity, we again refer to the surrogate frequency database to estimate the execution profile of a typical benign instance of each program used in the gadget. By adding interactions that mimic a benign process, we "camouflage" the gadget, dramatically improving the attack's evasion capabilities against graph-based ML detectors.



(a) Enterprise APT attack path instrumented with gadgets.

(b) Regularity Score calculation for causal paths for APT attack vs. APT attack with gadgets.

Figure 4.2: Change in regularity score due to gadget usage.

4.6.1 Evasive System Events (or Gadget) Finder

Certain program transitions in an attack chain can be conspicuous (*e.g.,* `excel.exe` executes `java.exe`). Intuitively, we would like to replace this conspicuous action with a more common one (*e.g.,* `excel.exe` executes `splwow64.exe`). If we make this choice naïvely, we may create additional conspicuous events later in the attack sequence. We must choose a replacement program that both avoids the conspicuous event and returns naturally to the original attack. In ProvNinja, such a program is called a "gadget". Unfortunately, there is almost never a single program that cleanly fits into the attack, so we extend the concept by chaining multiple gadgets together.

4.6.2 Applying Gadget Chains

With the goal of reducing the conspicuousness of our attack, we introduce the concept of a *gadget chain*: a sequence of events $(g_0, \dots, g_n)$ that will replace a conspicuous event e_k, such that $e_{k-1}.destination = g_0.source$ and $e_k.destination = g_n.destination$, allowing the gadget to naturally merge into the attack path. An effective gadget chain will improve the regularity of the attack by replacing rare events with more common ones, while still achieving the same end result.

We recursively search backward from the intended destination to the intended source to find gadget chains. We only include system events that have greater regularity than a user-defined threshold T, which is typically either an empirically chosen constant or a function of the regularity of the original event to be replaced (in our experiments we used empirically chosen constant $T = 0.03$ threshold). This formulation parameterizes the runtime and accuracy trade-off against exploring more gadget options. Notice that it is possible to fail to find any gadgets if the regularity threshold is too high. Finally, a domain expert chooses a gadget from the list to replace the conspicuous event in the attack path.

Table 4.1 shows a subset of the different gadget chains that can be used to replace a malicious event in the *establishing a foothold* stage in both the Enterprise and Supply-Chain APT scenarios. Table 4.2 provides an overview of different gadgets that can be utilized at various stages of enterprise and supply chain APTs. The five stages covered in this table include Initial Access, Establish a Foothold, Privilege Escalation, Deepen Access, and Exfiltration. These gadgets include common programs that are frequently used by attackers to gain access to systems, escalate privileges, and exfiltrate data. By understanding the types of gadgets used at each stage of an APT, the attacker can better prepare their attacks and increase their chance of a successful attack. Each stage requires a different set of tools and techniques, and understanding them can help defenders identify and prevent attacks at each stage. Furthermore, being able to detect an attack in its early stages can prevent an attacker from advancing to later stages and minimize the potential damage of an APT.

4.6.3 Camouflaging Gadgets

While the gadget chains improve the regularity of the attack path, any added processes are "naked" in that they only have events that are directly related to the attack; because

Table 4.1: Example gadgets with their normalized regularity score and problem space rejection reason. Regularity scores are normalized from 0 to 10, with a high score indicating higher regularity.

Index	Gadgets (Gadget Length)	Regularity Score	Rejection Rule
	`firefox.exe` **– (Gadgets) →** **`notepad.exe`**		
1	`svchost.exe` → **wininit.exe** → **winlogon.exe** → **userinit.exe** → `explorer.exe` (5)	2.8	Special Sequence
2	`svchost.exe` → **cmd.exe** → **shellexperiencehost.exe** (3)	8.3	Display Irregularities
3	**nssm.exe** → `python.exe` → `conhost.exe` → `wininit.exe` → `explorer.exe` (5)	4.39	Program Unavailability
4	**conhost.exe** → `werfault.exe` → `explorer.exe` (3)	8.1	Insufficient Privilege
5	`svchost.exe` → **schtasks.exe** → `conhost.exe` → `explorer.exe` (4)	7.9	Scheduling Tasks
6	`svchost.exe` → **rundll32.exe** → `winsat.exe` → `explorer.exe` (4)	9.1	Writing to Registries
7	**tvnserver.exe** → `mpcmdrun.exe` → `conhost.exe` → `explorer.exe` (4)	3.3	External Network Connections
8	**sshd.exe** → **ssh-shellhost.exe** → `explorer.exe` (3)	7.5	User Interactions
9	`sshd.exe` → **mpcmdrun.exe** → `conhost.exe` → `winword.exe` → `werfault.exe` → `explorer.exe` (6)	7.9	Singleton Programs
10	`services.exe` → **taskhostw.exe** → `ngentask.exe` → `ngen.exe` → `svchost.exe` → `explorer.exe` (6)	4.2	Special Protocol Support
11	`svchost.exe` → `werfault.exe` → `explorer.exe` (3)	9.5	-
	`python3` **– (Gadgets) →** **`wget`**		
12	`sh` → `perl` → **xfce-terminal** → `bash` (4)	3.9	Display Irregularities
13	`sh` → `bash` → **cargo** → `bash` (4)	4.4	Program Unavailability
14	`sh` → **anacron** → `sh` → `bash` (4)	3.1	Scheduling Tasks
15	`env` → **docker** → `bash` (3)	6.4	Resource Intensive Programs
16	`sh` → **nginx** → `bash` (3)	4.3	Configuration File Dependency
17	`sh` → **start-stop-daemon** → `sh` (3)	3.4	Network Disruption
18	`dash` → `bash` (1)	9.6	-

no intermediate actions are taken before creating the next process, the surrounding graph structure of a naked gadget is very distinct from that of a corresponding benign instance of the program. Graph-based provenance analysis models understand the surrounding graph

Table 4.2: APT attack stages first showing the original attack and then the attack using gadget chain along with their regularity score.

Attack Type	MITRE ATT&CK TTP	Gadgets	Reg. Score
Enterprise APT	Initial Access	`winlogon.exe` → `outlook.exe` → `explorer.exe` → `excel.exe`	1.3
	Establish a Foothold	`excel.exe` → `java.exe` → `x.x.x.x:443`	0.5
	Privilege Escalation	`excel.exe` → `java.exe` → `notepad.exe` → `x.x.x.x:445`	2.5
	Deepen Access	`java.exe` → `notepad.exe` → `cmd.exe` → `cscript.exe`	1.1
	Exfiltration	`cscript.exe` → `cmd.exe` → `sqlservr.exe` → `JDQKL.exe` → `osql.exe`	0.1
	Initial Access	`firefox.exe` → `svchost.exe` → `sdiagnhost.exe` → `services.exe` → `explorer.exe` → `notepad.exe`	7.6
	Establish a Foothold	`firefox.exe` → `svchost.exe` → `defrag.exe` → `werfault.exe` → `explorer.exe` → `notepad.exe` → `x.x.x.x:443`	6.5
	Privilege Escalation	`python.exe` → `conhost.exe` → `werfault.exe` → `explorer.exe` → `cmd.exe` → `x.x.x.x:445`	8.8
	Deepen Access	`cmd.exe` → `conhost.exe` → `werfault.exe`	6.7
	Exfiltration	`notepad.exe` → `werfault.exe` → `explorer.exe` → `firefox.exe`	9.1
	Initial Access	`firefox.exe` → `werfault.exe` → `explorer.exe` → `notepad.exe`	9.1
	Establish a Foothold	`cmd.exe` → `explorer.exe` → `svchost.exe` → `srtasks.exe` → `notepad.exe` → `x.x.x.x:443`	9.2
	Privilege Escalation	`python.exe` → `werfault.exe` → `winword.exe` → `firefox.exe` → `explorer.exe` → `cmd.exe` → `x.x.x.x:445`	8.5
	Deepen Access	`cmd.exe` → `explorer.exe` → `firefox.exe` → `svchost.exe` → `srtasks.exe` → `werfault.exe`	8.9
	Exfiltration	`notepad.exe` → `werfault.exe` → `explorer.exe` → `cmd.exe` → `services.exe` → `runtimebroker.exe` → `firefox.exe`	7.7
	Initial Access	`firefox.exe` → `svchost.exe` → `dstokenclean.exe` → `notepad.exe`	6.6
	Establish a Foothold	`cmd.exe` → `svchost.exe` → `disksnapshot.exe` → `werfault.exe` → `explorer.exe` → `notepad.exe` → `x.x.x.x:443`	7.3
	Privilege Escalation	`notepad.exe` → `firefox.exe` → `svchost.exe` → `python.exe` → `x.x.x.x:445`	9.2
	Deepen Access	`python.exe` → `conhost.exe` → `wininit.exe` → `werfault.exe`	6.0
	Exfiltration	`notepad.exe` → `explorer.exe` → `schtasks.exe` → `services.exe` → `dllhost.exe` → `runtimebroker.exe` → `firefox.exe`	7.8
Supply Chain APT	Initial Access	`bash` → `git` → `bash` → `docker`	6.5
	Establish a Foothold	`bash` → `sudo` → `docker` → `mount`	7.3
	Privilege Escalation	`bash` → `sudo` → `docker`	3.5
	Deepen Access	`docker` → `bash` → `python` → `bash` → `nmap`	2.1
	Exfiltration	`docker` → `bash` → `python` → `wget`	2.8
	Initial Access	`python` → `sh` → `start-stop-daemon`	9.8
	Establish a Foothold	`start-stop-daemon` → `bash`	7.4
	Privilege Escalation	`bash` → `dhclient3`	8.5
	Deepen Access	`dhclient3` → `bash` → `rsync`	5.8
	Exfiltration	`bash` → `curl`	7.5
	Initial Access	`python` → `sh` → `thunderbird`	6.2
	Establish a Foothold	`thunderbird` → `bash`	4.6
	Privilege Escalation	`bash` → `dbus-daemon`	9.8
	Deepen Access	`dbus-daemon` → `bash` → `firefox`	7.3
	Exfiltration	`bash` → `curl`	7.5
	Initial Access	`python` → `sh` → `bash` → `logrotate`	6.1
	Establish a Foothold	`logrotate` → `bash`	6.5
	Privilege Escalation	`bash` → `ntpd`	9.6
	Deepen Access	`ntpd` → `bash` → `scp`	8.9
	Exfiltration	`bash` → `curl`	7.5

structure, so they are easily able to detect the structural anomalies introduced by naked gadgets.

To mimic the graph structure of a benign program instance, we add events to the gadget based on the surrogate frequency database in our threat model. For each program p_g in the gadget, we divide the total number of each kind of resource interaction (*e.g.,* $(p_g, *, write), (p_g, *, read)$) by the total number of instances of p_g to estimate the typical distribution of entity interactions for a benign execution. We then add edges to the adversarial graph by sampling interaction targets for p_g from the surrogate frequency database. By keeping the distribution of events in our malicious instance similar to the distribution of events of the benign instances, our GNN-oriented ProvNinja implementation makes it difficult for GNNs to detect the use of gadget chains.

4.7 Problem Space Evasion

In this section, we discuss the challenges of implementing evasive attacks in the problem space. When considering a list of candidate attacks from the feature space, realizing them in the problem space becomes difficult due to complex system activity dynamics and environmental dependencies. Unlike in other ML domains, such as image processing, where the problem space closely resembles the feature space, system actions experience multiple transformations between data collection and feature embedding. Moreover, the problem space realization can be affected by the system environment as programs interact with other system components. The same system action executed on different systems, or even the same system at different points in time, may generate different provenance graphs.

After overviewing the principles suggested by Pierazzi et al.(Pierazzi et al., 2020), we present a set of filter rules that we specifically developed for realizing evasive attacks in the context of system provenance research. Although our current collection of rules is comprehensive, our system's design allows for the integration of further heuristics to minimize manual efforts and increase evasiveness.

4.7.1 Problem Space Constraints

Pierazzi et al.(Pierazzi et al., 2020) have extensively studied practical challenges related to problem space realization across various domains. We apply their systematic framework, which consists of four constraints, to analyze the problem space realization of ProvNinja in evading provenance-based ML detectors.

1. Available transformations. We use the event history in the frequency database to generate feature space attacks, as it indicates their previous occurrences and availability. However, when using public datasets as surrogate references for black-box attacks, discrepancies in available gadgets may occur. To address this, we actively prefer system programs §4.7.2, as they have a higher likelihood of availability.

2. Preserving attack semantics. Given a list of candidate system actions, expert knowledge and advanced skills are necessary to determine which ones preserve the semantics of the original attack. Although we can suggest a principled approach to (semi-) automate the process, the approach would involve numerous domain-specific considerations. In this work, we manually choose candidate system actions and verify their attack semantics equivalency, leaving the task of automated verification for future work.

3. Robustness to pre-processing. Unlike domains such as image or audio processing research with numerous transformative filters, system provenance datasets do not have a specific pre-processing stage influencing prediction results. However, we can still explore a line of data reduction research that proposes forensic-aware, lossy graph compression approaches to address storage and data processing pressures (Fei et al., 2021; Xu et al., 2016;

Tang et al., 2018). Assessing the impact of these data reduction schemes on the effectiveness of evasive attacks renders a promising research direction for future work (Inam et al., 2023).

4. Plausibility to users and security analysts. The newly constructed attack chain in the feature space should be *plausible* to regular users or security analysts. Furthermore, the attack must be unintrusive from user operations or system resource usage standpoints. Although manual investigations are still required, we preliminarily measure the number of nodes and edges added by ProvNinja's evasive actions in Figure 4.3. Limiting the event footprint induced by the attack reduces the chance that a user will notice the additional utilization of their system. We then filter out potentially intrusive actions using the automated rule set, as shown in §4.7.2.

4.7.2 System Provenance Filter Rules

To address the practical challenges of implementing problem space attacks, we developed gadget filters (*e.g.,* rejection rules) to minimize manual effort based on the following principles: *(1)* avoiding programs with large footprints and disruptions to users, *(2)* enforcing invariant rules associated with program execution sequences and permission levels, *(3)* the problem space should not make unnecessary modifications to the target host that would result in long-term or short-term side effects, and *(4)* prohibiting the use of black-listed programs (*e.g.,* `notepad.exe`) or suspicious behaviors (*e.g.,* registry updates to schedule background tasks or inject libraries).

While we suggest a comprehensive set of filter rules, our system design remains open to accommodating additional heuristics for automating the evasive attack generation process and enhancing their stealthiness. In §4.8.7, we evaluate their effectiveness in reducing the required manual effort.

Disturbances to End User and System Monitors

GUI interruptions. Some programs can be visually intrusive to be highly suspicious to users, such as a command prompt flashing on the screen or the file explorer opening. Therefore, gadget chains that include `cmd.exe` are rejected because a command prompt flashing on the screen will alert the user (*e.g.,* gadget path 2,12 in Table 4.1). Gadget paths including `explorer.exe` are not automatically excluded because it can be launched in the background with certain arguments.

Resource intensive programs. When a resource-intensive program (*e.g.,* `docker`) is run, it tends to draw more attention from the user and/or alert system monitors thus considered to be undesirable as shown as gadget chain 15 in Table 4.1.

External network connections and disruption. Programs that impersonate external socket connections can trigger network alerts and add overhead for attackers needing to set up receiving servers for camouflaging network reads. Gadget chain 7, 15 in Table 4.1 is rejected since `tvnserver.exe` connects externally to manage GPS data. Restarting networking processes, as in gadget chain 17 in Table 4.1, can indicate APT attacks, trigger security alerts, and lead to system instability or data loss.

Program-Specific Considerations

Configuration and operational dependencies. Programs that requires specific configuration or dependencies to other programs (*e.g.,* Web servers, `ngnix` that requires database access) would impose practical challenges as modifications to configuration files might require noticeable amount of modifications to the configuration that would result in abnormal system behavior. Standalone programs with less dependencies other applications and simple configuration are easier to control the program's runtime behavior.

Insufficient privilege. Certain gadgets require elevated privileges (*e.g.,* `NT_SYSTEM_SYSTEM`) to function. We analyze permission level consistency throughout the attack chain during the problem space realization, excluding gadgets that requires privilege escalation. For instance, the attack construction process rejects gadget path 4 in Table 4.1 as it mandates admin permissions to camouflage `conhost.exe`.

Special program sequences. Certain gadgets necessitate a specific position in the attack chain. For instance, `wininit.exe` is the first user program that initializes the userland applications followed by `winlogon.exe` and `userinit.exe` subsequently executes system programs such as `svchost.exe, conhost.exe,` and `nssm.exe`. While these special sequences are well-represented in the benign execution profiles of these system programs, they would appear highly suspicious during normal execution outside of the system bootstrap. Gadget path 1 in Table 4.1, for example, is automatically rejected because it contains a special sequence.

Blacklisted Programs and Suspicious Behaviors

Blacklisted programs. Certain programs are under high scrutiny based on the fact that those programs have been historically hijacked or impersonated by malware. We can subscribe to Cyber Threat Intelligence (CTI) feeds for the up-to-date blacklist to reject suspicious gadgets.

Modification to system resources. Efforts should be made to actively reject gadgets that modify sensitive system resources such as libraries for payload objectives, as adding camouflage may inadvertently link distinct system programs' provenance graphs together due to information flow. For example, gadget chain 6, 15 in Table 4.1 is excluded because camouflaging `services.exe` involves writing to system libraries (`KernelBase.dll.mui`, `ntdll.dll`) which are also read by `nssm.exe`, connecting the two graphs.

Modification to system configuration. Attackers often modify system configurations (*e.g.,* Windows registry, Linux crontab, and RC files) to plant malicious activities such as scheduling malware execution or interposing library loading. Since the security community is well-aware of these practices, we prevent such sensitive operations from being included in the attack chain. Gadget chain 6 in Table 4.1 is rejected because `rundll32.exe` writes to registries. Recent studies on Fileless malware reveal a preference for system programs, as they typically consume substantial resources in daily usage. Gadget chain 5, 14 in Table 4.1, which is also rejected, displays a process attempting to schedule a task, as it requires calling `schtasks.exe`.

Surrogate Model Discrepancies

The limitations of publicly available datasets used for building surrogate models can result in an inadequate representation of victim networks. Such datasets may feature programs and operational behaviors specific to their data collection. For example, the reference dataset in our study contains obsolete programs like `nssm.exe`, discontinued after 2016 (nss, 2021), and proprietary programs like `cargo`, Rust's package manager. Using these programs to create gadget chains, such as 3, 13 in Table 4.1, leads to unrealizable outcomes in the defender's system due to their unavailability.

4.8 Evaluation

In this section, we extensively evaluate the effectiveness of ProvNinja at evading provenance-based IDS. Our evaluation aims to answer the following research questions:

- **RQ1: Feature Space Evasion.** Do ProvNinja's evasive attacks effectively evade ML-based detectors (§4.8.5) under different threat models (§4.8.6)?
- **RQ2: Problem Space Attack Realization.** Can ProvNinja's evasive attacks be realized in the problem space (§4.8.7)?
- **RQ3: Surrogate Data Effectiveness.** How effective is surrogate data in generating evasive attacks? (§4.8.8)

- **RQ4: Attack Transferability.** How well do ProvNinja's evasive attacks transfer between defense model architectures? (§4.8.9)

4.8.1 Evaluation Methodology

We evaluate ProvNinja's ability to generate evasive attack sequences against provenance-based IDS using our enterprise and supply chain APTs §2.9, and Fileless malware Table 2.8 attack scenarios. In our evaluation, the defense models are trained on our large benign dataset that includes 13 months of organic user activity. The attacker uses the publicly available DARPA Transparent Computing dataset (DARPA, 2019) as a surrogate dataset following the blackbox threat model (§4.2). To comprehensively explore ProvNinja's effectiveness on Fileless malware, we refer to existing work (Barr-Smith et al., 2021) and collect samples from a popular malware repository (VirusTotal, 2021). We ran 5,925 Fileless malware samples, categorized them by the system programs they impersonated, and used the 10 largest categories in our evaluation (refer to Table 2.2).

We first evaluate the evasiveness of ProvNinja in the feature space, then we show that these attack chains are realizable in the problem space. We specifically measure the effectiveness of ProvNinja in evading four prominent ML detectors from the literature, which employ two distinct embedding approaches: (1) ProvNinja-PATH targeting path-embedding detectors, such as ProvDetector and SIGL , and (2) ProvNinja-GRAPH focusing on graph-embedding detectors, like GAT and Prov-GAT . Additionally, we evaluate ShadeWatcher (Zengy et al., 2022), the SOTA GNN-based anomaly detector for system provenance for the general applicability of ProvNinja approach.We select evasive attack options that are within our capability to implement, then leverage industry-grade pen-testing tools (Metasploit, 2021; MetasploitVenom, 2021) to create an evasive attack which can be launched against real systems.

Table 4.3: presents the detection results for the baseline, randomly perturbed, and Ninja attacks, with a lower F-1 score indicating better evasion. We replace rare edges with a random sequence of programs in random perturbations and with evasive gadgets in Ninja attacks. We display differences from the baseline values inside parentheses.

Attack Type	GNN-Based Detectors	Baseline			Random perturbation		ProvNinja-GRAPH	
		Precision	Recall	F1	Recall	F1	Recall	F1
Enterprise APT	GAT	0.94	0.74	0.83	0.69 *(-.05)*	0.78 *(-.05)*	0.37 *(-.37)*	0.54 *(-.29)*
Supply Chain APT		0.93	0.78	0.85	0.96 *(+.18)*	0.91 *(+.06)*	0.44 *(-.34)*	0.53 *(-.32)*
Fileless Malware		0.95	0.94	0.95	0.92 *(-.02)*	0.94 *(-.01)*	0.71 *(-.23)*	0.77 *(-.18)*
Enterprise APT	Prov-GAT	0.95	0.95	0.95	0.71 *(-.24)*	0.68 *(-.27)*	0.25 *(-.70)*	0.37 *(-.58)*
Supply Chain APT		0.94	0.96	0.95	0.85 *(-.11)*	0.90 *(-.05)*	0.28 *(-.68)*	0.56 *(-.39)*
Fileless Malware		0.96	0.98	0.97	0.96 *(-.02)*	0.96 *(-.01)*	0.58 *(-.40)*	0.67 *(-.30)*
Average		**0.95**	**0.90**	**0.92**	**0.85 *(-.05)***	**0.86 *(-.06)***	**0.44 *(-.46)***	**0.57 *(-.35)***

Attack Type	Path-based Detectors	Baseline			Random perturbation		ProvNinja-PATH	
		Precision	Recall	F1	Recall	F1	Recall	F1
Enterprise APT	ProvDetector	0.98	0.78	0.87	0.88 *(+.10)*	0.92 *(+.05)*	0.23 *(-.55)*	0.31 *(-.56)*
Supply Chain APT		0.99	0.92	0.90	0.95 *(+.03)*	0.95 *(+.05)*	0.35 *(-.57)*	0.30 *(-.60)*
Fileless Malware		0.91	0.91	0.91	0.94 *(+.03)*	0.93 *(+.02)*	0.33 *(-.58)*	0.43 *(-.48)*
Enterprise APT	SIGL	0.97	0.99	0.98	0.99 *(+.00)*	0.99 *(+.01)*	0.30 *(-.69)*	0.41 *(-.57)*
Supply Chain APT		0.90	0.90	0.90	0.96 *(+.06)*	0.95 *(+.05)*	0.38 *(-.52)*	0.43 *(-.47)*
Fileless Malware		0.91	0.95	0.93	0.98 *(+.03)*	0.99 *(+.06)*	0.47 *(-.48)*	0.57 *(-.36)*
Average		**0.94**	**0.91**	**0.92**	**0.95 *(+.04)***	**0.96 *(+.04)***	**0.34 *(-.57)***	**0.41 *(-.51)***

4.8.2 Evaluation Datasets

Benign dataset. With the approval and oversight of our university's Institutional Review Board (IRB), we solicited written informed consent from volunteers to participate in a long-running provenance data collection project. Using Linux kernel audits and Windows ETW event tracing, we collected provenance data involving file, process, and network events. Our volunteers performed a variety of workloads as students, researchers, developers, and administrators. In aggregate, our volunteers have helped us collect system event data from 54 Windows hosts and 32 Linux hosts over 13 months, yielding 17TB of system event data for our benign dataset.

DARPA transparent computing (TC) dataset. The DARPA Transparent Computing Engagement 3 and 5 Data Releases (DARPA, 2019) include extensive system logs of both benign and malicious activities, which can be used to generate a surrogate frequency database. However, this dataset is notably limited due to the short duration of the engagements and the scripted nature of the captured activities. The primary challenge for its use in ProvNinja lies in the limited selection of user/internet-facing applications that execute system programs, which restricts ProvNinja's flexibility near the point of entry. We utilized E3/5 Theia and Trace, E3/5 FiveDirections, and E5 Marple for Linux and Windows gadget mining.

Enterprise APT. We ran the enterprise APT attack campaign (§2.9) on a local testbed environment which consisted of four windows and three Linux hosts. The recorded system event logs constitute our Enterprise APT dataset. We then generated provenance graphs for the programs used in key stages of the enterprise APT scenario: `excel.exe`, `java.exe`, `notepad.exe`, `osql.exe`, `explorer.exe`, and `outlook.exe`. This collection procedure yielded 1,779 provenance graphs with an average of ~176 causal paths for each graph.

Because provenance graphs include unpredictable background system interactions that can affect the performance of the models, we ran the scenario multiple times to sample the distribution of noise in the system and show that ProvNinja's evasive attacks are effective in real-world conditions.

Supply Chain APT. We ran the Supply-Chain APT campaign on a local Linux test bed which is consisted of five Linux hosts. We generated provenance graphs for `python`, `curl`, `docker`, `git`, `thunderbird`, and `firefox` and the recorded system event logs constitute our Supply-Chain APT dataset. This collection procedure yielded 1,091 provenance graphs with an average of ~494 causal paths each.

Fileless malware. Leveraging a public dataset of Fileless malware (Barr-Smith et al., 2021), we collected and ran 5,925 malware samples on our distributed Cuckoo (cuc, 2019) sandbox environment. We collected provenance graphs from each sample to capture all the triggered malicious behaviors. Then, we select the top 10 most well represented impersonation targets (summarized in Table 2.2) and their graphs to include in our Fileless malware dataset. In total, our Fileless Malware dataset consists of 1,206 high-quality provenance graphs. This dataset characterizes the runtime behavior of Fileless malware, opening these sophisticated techniques for further analysis.

Experimental bias in malware analysis. Kuchler et al.(Küchler et al., 2021) and Avllazagaj et al.(Avllazagaj et al., 2021) have emphasized the importance of considering experimental bias in malware analysis. The use of virtual environments like the Cuckoo sandbox (cuc, 2019) can introduce biases due to differences in trigger conditions and freshness, which can significantly affect malware behavior compared to the behavior of malware *in the wild*.

One of the main challenges associated with experimental bias is the potential for certain types of malware to be selectively chosen for analysis. For example, if samples are selected based on activity or freshness, there may be a bias towards highly active or prevalent malware, while less prevalent or subtle types of malware may be overlooked. To mitigate this issue, we manually verified malicious behavior in 1206 of our 5925 samples, prioritizing system programs less affected by custom configurations and user interactions, such as `rundll32.exe`.

Furthermore, downloading malware samples from sites such as VirusTotal (VirusTotal, 2021) can also introduce bias into the dataset. For example, antivirus programs may have already classified the samples, potentially skewing the results of any subsequent analysis. Additionally, the samples themselves may not be representative of the overall population of malware, as they may be biased towards the types of malware more commonly detected by antivirus programs. To minimize potential bias, we carefully selected our dataset by meticulously reviewing threat reports generated by (VirusTotal, 2021) to better understand the malware's behavior.

Overall, the realism of the malware experiments is constrained by: *(1)* the sandbox execution environment, which will not capture the behavior of malware equipped with sophisticated sandbox detection mechanisms, a concern highlighted by (Küchler et al., 2021); *(2)* the prioritization of system programs that are not sensitive to user activity or configuration changes, which reduces the variance in the captured behaviors at the cost of underrepresenting user-facing programs, as noted by (Avllazagaj et al., 2021); *(3)* the use of online malware repositories, which overrepresents detectable malware instances. The challenge of accurately representing and profiling the full malware landscape remains an open and orthogonal problem.

4.8.3 Dataset Statistics

Benign Program Profiles. In this section, we provide detailed statistics on the system provenance graphs used throughout this dissertation to evaluate ProvNinja. We selected 52 system programs from our event database that are commonly used in APT campaigns from previous studies (Wang et al., 2020; Liu, Zhang, Li, Jee, Li, Wu, Rhee, and Mittal, Liu et al.; Han et al., 2021; Cozzi et al., 2018b). The list can be found in Table 5.4. The program list consists of two kinds of programs: system programs used by the OS for system functionalities and user programs that are used in everyday general workloads. On average, the provenance graphs generated from the benign system programs contained 4,735.30 causal paths, 37.51 vertices and 45.78 edges on average (Table 5.4). The provenance graph generated from the benign user application consisted of 11,779.36 causal paths, 90.36 vertices, and 112.38 edges on average (Table 5.4).

Malicious Dataset. There are three anomalous datasets: Enterprise APT, Supply-Chain APT, and Fileless Malware. We conducted our experiment for each of the APT attack stages (*e.g., Initial Access*, *Establish a Foothold*, *Privilege Escalation*, *Deepen Access* and *Exfiltration*). The provenance graphs for *Enterprise APT* contain an average of 493.92

Table 4.4: Benign graph size for system programs.

Applications	Avg # of causal paths	Avg # of total vertices and edges	Avg # of forward vertices and edges	Avg # of backward vertices and edges
		System Programs		
		Windows		
`acrord32.exe`	1957.08	39.58 / 46.51	6.28 / 5.28	33.3 / 41.23
`certutil.exe`	28162.32	35.09 / 74.54	3.37 / 2.37	31.72 / 72.17
`cmd.exe`	2462.71	21.99 / 26.71	5.57 / 4.57	16.42 / 22.14
`code.exe`	10579.16	67.06 / 92.53	16.53 / 15.53	50.53 / 77.0
`conhost.exe`	4418.39	33.92 / 35.51	2.01 / 1.01	31.91 / 34.5
`cscript.exe`	6949.2	52.8 / 65.2	2.0 / 1.0	50.8 / 64.2
`cvtres.exe`	24.5	11.5 / 10.0	2.0 / 1.0	9.5 / 9.0
`msiexec.exe`	11473.0	74.0 / 96.0	2.0 / 1.0	72.0 / 95.0
`netsh.exe`	4181.39	34.18 / 44.14	2.31 / 1.31	31.87 / 42.83
`powershell.exe`	1429.78	33.28 / 38.69	5.06 / 4.06	28.22 / 34.63
`sc.exe`	270.05	10.06 / 9.31	2.89 / 1.89	7.17 / 7.42
`svchost.exe`	4.54	5.62 / 3.62	3.31 / 2.31	2.31 / 1.31
`tasklist.exe`	123.0	14.33 / 19.67	2.0 / 1.0	12.33 / 18.67
`taskmgr.exe`	3621.88	42.83 / 50.33	2.0 / 1.0	40.83 / 49.33
`userinit.exe`	77.0	89.34 / 87.34	86.67 / 85.67	2.67 / 1.67
`winlogon.exe`	30.75	34.5 / 32.5	31.0 / 30.0	3.5 / 2.5
		Linux		
`dash`	153808.57	371.87 / 381.97	211.61 / 206.44	160.26 / 175.53
`dd`	213601.29	995.5 / 1003.6	551.68 / 501.81	443.82 / 501.79
`ps`	181846.43	834.01 / 998.14	369.21 / 501.77	464.8 / 496.37
`sh`	208367.43	445.01 / 851.27	4.16 / 357.78	440.85 / 493.49
`smbd`	201559.57	355.37 / 371.15	9.69 / 3.39	345.68 / 367.76
`sshd`	182601.57	233.04 / 234.15	9.35 / 6.6	223.69 / 227.55
`bash`	166355.43	454.25 / 510.76	10.57 / 9.31	443.68 / 501.45
`cron`	214827.71	327.16 / 241.85	10.27 / 9.96	316.89 / 231.89
`cat`	184346.43	310.51 / 210.9	9.0 / 6.99	301.51 / 203.91
`dbus-daemon`	156713.0	20.16 / 20.04	9.02 / 6.42	11.14 / 13.62
`ls`	179185.86	213.62 / 356.47	10.25 / 9.3	203.37 / 347.17
`perl`	809.0	25.01 / 23.22	11.95 / 12.05	13.06 / 11.17
`rm`	174590.43	452.89 / 440.38	15.06 / 18.5	437.83 / 421.88
`cp`	175636.86	193.42 / 212.7	179.09 / 184.69	14.33 / 28.01
`grep`	212413.86	191.51 / 502.32	13.51 / 16.43	178.0 / 485.89
`service`	231.43	18.32 / 21.24	15.32 / 18.55	3.0 / 2.69
Average	**4735.30**	**37.51 / 45.78**	**10.94 / 9.93**	**26.57 / 35.85**
		User Programs		
		Windows		
`acrobat.exe`	92.08	11.35 / 14.32	2.46 / 1.46	8.89 / 12.86
`chrome.exe`	3028.15	50.17 / 58.69	2.01 / 1.01	48.16 / 57.68
`discord.exe`	2228.39	61.42 / 76.29	26.03 / 25.03	35.39 / 51.26
`excel.exe`	33113.53	131.54 / 158.53	35.67 / 34.60	95.87 / 123.93
`explorer.exe`	9119.99	327.03 / 371.69	315.41 / 355.87	11.62 / 15.82
`firefox.exe`	9792.64	78.66 / 91.53	21.15 / 20.44	57.51 / 71.09
`javaw.exe`	22500.40	71.82 / 123.45	10.58 / 20.76	61.24 / 102.69
`notepad.exe`	34141.24	92.07 / 144.66	2.22 / 1.19	89.85 / 143.47
`osql.exe`	415.29	26.15 / 32.29	3.29 / 2.29	22.86 / 30.0
`outlook.exe`	42796.90	219.80 / 267.90	90.00 / 88.90	129.80 / 179.00
`pycharm64.exe`	850.38	28.51 / 31.13	7.02 / 6.33	21.49 / 24.8
`python.exe`	248.67	10.93 / 11.17	3.0 / 2.0	7.93 / 9.17
`slack.exe`	3242.43	96.71 / 119.57	30.57 / 29.57	66.14 / 90.0
`word.exe`	3341.0	58.88 / 72.0	26.38 / 25.38	32.5 / 46.62
		Linux		
`java`	169180.71	133.94 / 222.4	17.44 / 19.63	116.5 / 202.77
`python`	161755.57	365.71 / 348.31	11.51 / 8.14	354.2 / 340.17
`firefox`	176843.86	194.22 / 504.56	15.84 / 18.78	178.38 / 485.78
`nginx`	258367.17	514.27 / 514.13	500.76 / 501.26	13.51 / 12.87
`git`	231.43	18.32 / 21.24	15.32 / 18.55	3.0 / 2.69
`docker`	9113.02	513.80 / 510.54	501.67 / 497.45	12.13 / 13.09
Average	**11779.36**	**90.36 / 112.38**	**41.13 / 43.92**	**49.23 / 68.46**

causal paths, 94.78 vertices, and 97.48 edges. The provenance graphs for *Supply-Chain APT* have an average of 175.93 causal paths, 30.39 vertices and 29.50 edges. The provenance graphs for *Fileless Malware* contain an average of 4302.05 causal paths, 177.75 vertices, and 211.96 edges.

4.8.4 Baseline Performance of ML Detectors

To measure the baseline performance of the four different detection models, we tested their performance against the enterprise and supply chain APT scenarios, as well as our collection of Fileless malware. The results are summarized in the baseline columns of Table 4.3. Overall, provenance-based ML detectors have provided practical defense. The GNN-based detectors have shown 0.95, 0.90 and 0.92 for precision, recall and F1 score, whereas path-based detectors have shown 0.94, 0.91 and 0.92 for precision, recall and F1 score for their baselines.

The GAT model's average recall and F1 scores across our test cases are 0.82 and 0.88. Notably, the GAT performs relatively poorly against the enterprise and supply chain APTs. The structure of the APT graphs is similar to that of benign graphs, so the structure-only GAT struggles to accurately classify this attack. The Prov-GAT model's average recall and F1 scores across our test cases are 0.96 and 0.96, respectively. Because Prov-GAT sees node attributes (*e.g.,* file names, ip addresses, *etc.*), it leverages this information to perform more accurate classification. Prov-GAT performs well in all of our categories, demonstrating that the model is able to take advantage of the additional node attribute information.

The ProvDetector model's average recall and F1 scores across our test cases are 0.87 and 0.89. Notably, the ProvDetector model performs poorly against the enterprise APT, similar to the GAT . ProvDetector is an anomaly detection layer on top of Doc2Vec (Le and Mikolov, 2014), so it has limited awareness of the structure of the causal path. The enterprise APT contains related programs that are relatively close in the neural embedding space compared to those of our other test cases. The SIGL model's average recall and F1 scores across our test cases are 0.95 and 0.93. This performance is comparable to that of the Prov-GAT model. Because SIGL internally learns to reconstruct the entire causal path, it has strong sensitivity to the context of programs in the causal path.

4.8.5 Feature Space Evasion

In this section, we evaluate the effectiveness of ProvNinja's suggested ninja attack chains at evading the detection models. Recall that our feature space modifications include the addition and replacement of nodes and edges (§4.6.2).

Random gadgets and camouflage. To demonstrate the robustness of the models to random changes in the attacks, we implemented a variant of our ProvNinja framework that makes random gadget and camouflage selections. The process of locating conspicuous edges is the same as in ProvNinja, but gadgets are chosen randomly from the list of available programs instead of intelligently choosing from the frequency database. Table 4.3 shows that the models still detect random variants of the attacks with high accuracy. The random modification scheme reduced the recall of the defense models by an average of 4.5% and reduced the F1 scores by an average of 5%.

ProvNinja-PATH effectiveness. Against the path-based models (ProvDetector and SIGL), ProvNinja-PATH devised 81 ninja variants of our Enterprise APT, 55 ninja variants of our supply chain APT, and ninja variants of our Fileless malware collection. ProvNinja-PATH reduced the average recall and F1 for ProvDetector and SIGL by 57% and 51%, respectively.

ProvNinja-GRAPH effectiveness. Against the graph-based (GAT and Prov-GAT), ProvNinja-GRAPH devised 47 ninja variants of our enterprise APT and 28 ninja variants of our supply chain APT, as well as ninja variants for our Fileless malware collection. Using the surrogate frequency dataset, ProvNinja-GRAPH was able to identify and modify

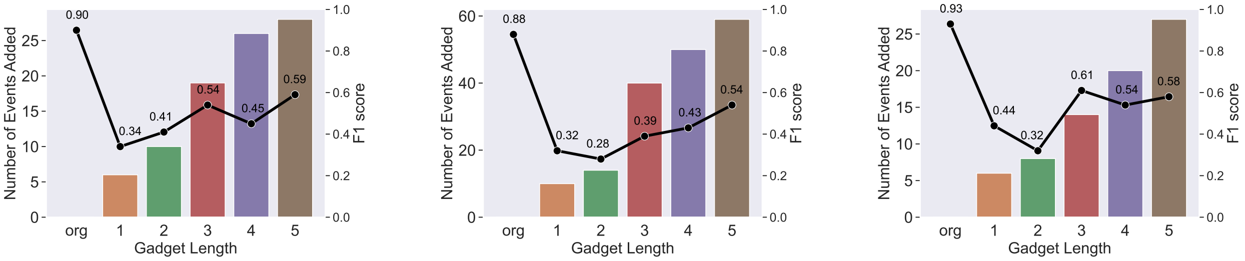
Table 4.5: ProvNinja evasion for ShadeWatcher (Zengy et al., 2022).

Attack Type	ShadeWatcher		Random Perturb.		ProvNinja	
	Recall	F1	Recall	F1	Recall	F1
Enterprise APT	0.96	0.93	0.98*(+.02)*	0.98*(+.05)*	0.45*(-.51)*	0.41*(-.52)*
Supply Chain APT	0.92	0.90	0.96*(+.04)*	0.97*(+.07)*	0.38*(-.54)*	0.40*(-.50)*
Average	**0.94**	**0.92**	**0.97*(+.03)***	**0.98*(+.06)***	**0.42*(-.53)***	**0.41*(-.51)***

conspicuous edges that contributed heavily to the detection of the attack. Against the GAT and Prov-GAT models, the ninja attack variants reduced the average recall and F1 scores by 46% and 35%.

Side effects. Gadgets camouflaged with additional events inevitably introduce *side effects* (noise) which we measured through differences in graph size compared to the original attack graph. In Figure 4.3 we see that using longer gadget chains results in more noise in the provenance graph, as well as worse performance. Long gadgets require additional engineering effort to craft, increase the number of points of failure, and tend to perform worse than short gadgets. Therefore, we prefer shorter gadgets since the potential for unintended side-effect (*e.g.,* noise) is reduced. Also, it is critical to make informed choices about which edges to add to minimize the chance of detection.

ProvNinja evasion for ShadeWatcher (Zengy et al., 2022). In addition to four provenance-based ML detectors, we also conducted a comparison study using SOTA GNN-based IDS ShadeWatcher (Zengy et al., 2022), which is based on recommendation systems. The results in Table 4.5 show that ShadeWatcher's recall and F1 score decrease significantly when using the ProvNinja approach compared to random perturbations. The APT variants produced by ProvNinja reduced detection of malicious activity, attributed to its ability to find benign transformations for malicious edges. This aligns with our expectations for ProvNinja, which can disguise anomalous graph instances as benign through edge-level augmentation. The success of ProvNinja against ShadeWatcher, specializing in fine-grained edge-level detection, demonstrates its capability to counter robust provenance-based ML detectors.



(a) Enterprise APT.

(b) Supply Chain APT.

(c) Fileless Malware.

Figure 4.3: Events added and F1 score vs gadget length. In each evaluation scenario, bars represent number of additional events whereas solid lines are for F-1 score trends.

This study compares the effectiveness of ShadeWatcher (Zengy et al., 2022) against enterprise and supply chain APT scenarios compared with ProvNinja's adversarial perturbations, and random perturbations. Table 4.5 shows that the recall and F1 score of ShadeWatcher against anomalous graphs generated by the Enterprise and Supply-Chain APTs decrease significantly when ProvNinja is used to mask the attacks, but stays the same when random perturbations are applied. On average, there is a 64% decrease in both recall and F1 score when ProvNinja is used, and 9% decrease in recall and 6% decrease in F1 score when random semantically equivalent perturbations are done.

The findings indicate that the APT variants produced by ProvNinja exhibit significantly reduced detection of malicious activity compared to the original APT attacks. We attribute this outcome to the ability of ProvNinja to find equivalent benign transformations for malicious edges, effectively hiding them from the IDS. This outcome is precisely what we expect from the application of ProvNinja, where anomalous graph instances can masquerade as benign through self-augmentation on an edge-level, reducing the visible attack surface instead of polluting the search graph. The success of ProvNinja against ShadeWatcher (which specializes in fine-grained, edge-level detection) confirms its ability to combat robust provenance graph-based IDSs.

Table 4.6: ProvNinja's performance under White-box, Black-box and Blind threat model, evaluated for two configurations of Blind (ProvNinja) and Blind (random perturbation).

Defense Model	White-box (ProvNinja)		Black-box (ProvNinja)		Blind (ProvNinja)		Blind (Random Pert.)	
	Recall	F1	Recall	F1	Recall	F1	Recall	F1
ProvDetector	0.23	0.27	0.30 *(+.07)*	0.35 *(+.08)*	0.60 *(+.37)*	0.67 *(+.40)*	0.89 *(+.66)*	0.91 *(+.64)*
SIGL	0.31	0.35	0.38 *(+.07)*	0.47 *(+.12)*	0.69 *(+.38)*	0.74 *(+.39)*	0.97 *(+.66)*	0.95 *(+.60)*
GAT	0.38	0.41	0.42*(+.04)*	0.51 *(+.10)*	0.75 *(+.37)*	0.77 *(+.36)*	0.91 *(+.53)*	0.93 *(+.52)*
Prov-GAT	0.44	0.47	0.51 *(+.07)*	0.61 *(+.14)*	0.78 *(+.34)*	0.80 *(+.33)*	0.96 *(+.52)*	0.97 *(+.50)*
ShadeWatcher	0.36	0.33	0.42 *(+.06)*	0.41 *(+.08)*	0.75 *(+.39)*	0.72 *(+.39)*	0.97 *(+.61)*	0.97 *(+.64)*
Average	**0.34**	**0.37**	**0.41 *(+.06)***	**0.47 *(+.10)***	**0.71 *(+.37)***	**0.74 *(+.37)***	**0.94 *(+.60)***	**0.95 *(+.58)***

4.8.6 White-box and Blind Threat Models

We also evaluated ProvNinja, mainly implemented for a black-box threat model, under relaxed white-box and stricter blind threat models. To showcase its effectiveness in finding suitable replacement gadgets, we evaluate the blind threat model under two design choices regarding replacement gadget selection: with ProvNinja and with random perturbation.

In the white-box model, the attacker has complete access to the defender's model internals and data, enabling them to use a white-box GNNExplainer (Yuan et al., 2022) to supplement the regularity score when deciding which events to replace and to find the replacement gadgets. In the blind threat model, attackers have no prior access to the defender's environment, model, or data. Therefore, they rely on a public dataset to construct a surrogate model and identify rare events. We consider two types of blind attacks. The blind attack with ProvNinja constructs a model using the public dataset for appropriate gadget replacements, while the blind attack with random perturbation attempt makes random selections for its gadgets.

As shown in Table 4.6, recall and F1 scores are lower for weaker defense models across different threat models, which is expected since it is easier to create ninja variants against weaker defense models. Our results demonstrate that SIGL and Prov-GAT achieve higher recall and F1 scores compared to that of ProvDetector and GAT for different threat models.

Also, not surprisingly, we notice that the performance of the ninja variants improves when they are tested against defense models where the attacker's model closely approximate the defender's model.

We observe in Table 4.6 that the difference between black-box and white-box in terms of recall and F1 scores is 6% and 10%, respectively, while the difference between blind threat model that uses ProvNinja and white-box is 37% for both recall and F1 score. The larger difference between blind where ProvNinja was used and white-box is attributed to the varying workloads between the surrogate and the defender's dataset. Therefore, the successful evasion of the surrogate model does not guarantee the evasion of the defender's model, as most of the evasive gadgets created from the surrogate data did not transfer over to the defender's model.

However, ProvNinja retains partial effectiveness even under the blind threat model. The blind attack that does not use ProvNinja (*i.e.,* random perturbation attack) had a 60% increase in recall and a 58% increase in F1 score compared to the white-box attack, which is worse than ProvNinja's 37% increase in both recall and F1 score under the same threat model. We note that the attacker's surrogate model was trained on the publicly available but limited engagement DARPA dataset; an attacker with more comprehensive provenance data could train a stronger surrogate model and achieve better blind performance. Despite this limitation, we consider even one successful evasive attack as a successful outcome for that APT stage, following the approach of other work (Goyal et al., 2023).

4.8.7 Problem Space Realization

To further refine the feature space of gadgets for implementing problem space evasion attacks, we employ the recommendations from §4.7. Moreover, we assess the amount of effort (in analyst person hours) necessary to execute a gadget chain.

Table 4.7 summarizes the filtration process of 211 feature space candidates according to the rules outlined in §4.7, resulting in 22 distinct variations for enterprise and supply chain

Table 4.7: Number of unviable candidates removed by each recommendation §4.7.2.

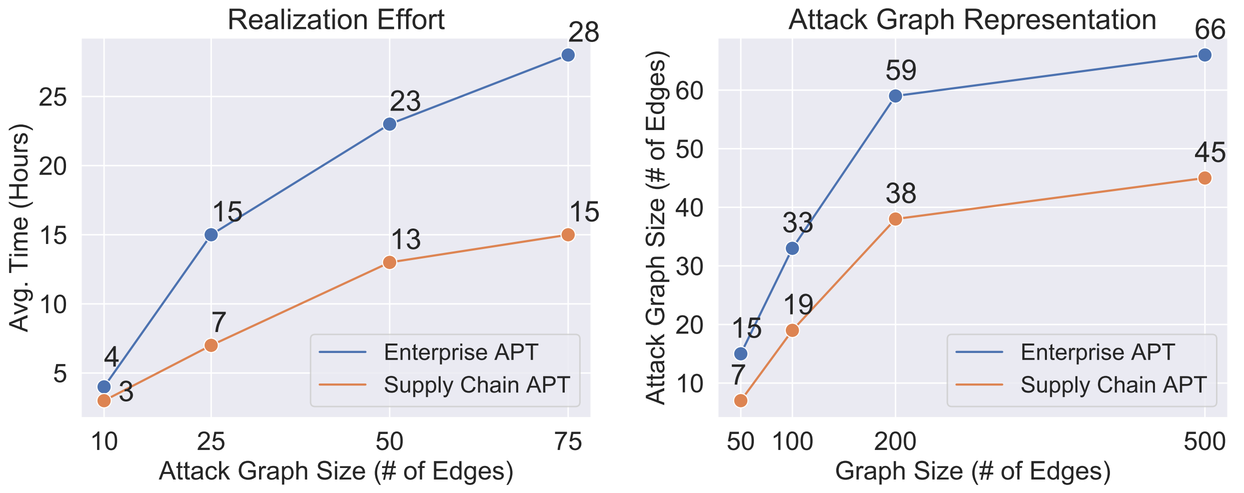
Attack Type	Feature Space Attacks	Disturbances to End User and System Monitors			Program-Specific Considerations		Blacklisted Programs and Suspicious Behaviors			Surrogate Model Discrepancies	Problem Space Attacks
		GUI interruptions	Resource intensive programs	External network conn. and disruption	Insufficient privilege	Special program sequences	Blacklisted programs	Modification to system resources	Modification to system configuration		
Enterprise APT	128	7	5	25	17	23	6	11	8	12	14
Supply Chain APT	83	12	13	10	3	9	2	14	7	5	8

APT scenarios. In the Enterprise APT scenario, 128 initial feature space candidates were reduced by 89% to 14 problem space attacks, with most discarded attack variants involving gadget chains requiring external network connections; these variants could have triggered the defender's firewall rules, raising unnecessary suspicion. In the Supply-Chain APT scenario, 83 initial feature space candidates were reduced by 90% to yield 8 problem space attacks, with program unavailability posing the greatest obstacle to attack realization.

Attack variant example that includes singleton programs: `firefox.exe` → `svchost.exe` → `wininit.exe` → `services.exe` → **sshd.exe** → `explorer.exe` → `notepad.exe`. This gadget chain aims to inconspicuously achieve privilege escalation through `notepad.exe` while starting at `firefox.exe`, but runs the risk of creating a new instance of the singleton program `sshd.exe`, which listens to sockets and manages remote system logins. Multiple concurrent instances of `sshd.exe` would be clearly abnormal, alerting the defender.

A gadgeted path whose objective is to execute `notepad.exe` (a privilege escalation malware) inconspicuously from the system's entry point, `firefox.exe` will include these gadgets: `firefox.exe` → `svchost.exe` → `wininit.exe` → `services.exe` → **sshd.exe** → `explorer.exe` → `notepad.exe`. In the gadget chain, `sshd.exe` is used but in a benign setting only single instance is allowed to run at a time since the program listens to sockets and manages remote system logins. This gadget chain runs the risk of having two instances of `sshd.exe` running if a remote user logs in, therefore, raising the suspicion among security experts.

An example where a gadget chain was discarded following the "external network" recommendation is: `firefox.exe` → **officeclicktorun.exe** → `schtasks.exe`→ `services.exe` →



(a) Time required to actualize the gadget for different attack graph.

(b) Composition of attack graphs in comparison to the whole graph.

Figure 4.4: Realization effort for larger graphs takes more time, but there is a diminishing result since the number of rare edges and gadgets are limited for a particular attack stage.

`msmpeng.exe` → `runtimebroker.exe` → `werfault.exe` → `explorer.exe` → `notepad.exe`. The program in question is **officeclicktorun.exe**. In a benign setting, it manages updates for Microsoft Office products, coordinates resources, and handles background streaming. Checking for and deploying updates result in a significant number of external network reads, which must be mimicked to properly impersonate `officeclicktorun.exe`.

Manual efforts for ProvNinja evasion. We actively analyzed the effort required to implement PROVNINJA's evasive attacks, estimating it in terms of security analysts' hours and taking graph size into account as a key factor. This effort encompasses tasks such as: *(1)* running PROVNINJA to obtain filtered feature space gadgets; *(2)* meticulously evaluating problem space recommendations to discard evasive gadgets from the filtered list of feature space gadgets; *(3)* selecting gadgets for implementation with various pentesting frameworks; and *(4)* implementing the selected gadgets in the problem space. Results in Figure 4.4a reveal that as attack graphs grow larger, implementing evasive attacks becomes increasingly time-consuming.

Interestingly, as shown in Figure 4.4b, we found that the implementation effort is sublinear in the size of the attack graph. Since the majority of system events are benign, the attack graph's size is also sublinear in the total graph size. When comparing the Enterprise APT scenario, we discovered that implementing the Supply Chain APT gadgets takes less time. This is attributable to the numerous replacement options available from surrogate datasets that closely resemble defender datasets (illustrated in §4.8.8). This similarity enables the creation of many gadgets using the surrogate dataset on the defender model.

We have analyzed the estimated effort required to implement ProvNinja's evasive attacks in terms of domain experts' hours, taking into account the graph size as a key factor. The domain experts' hours include several tasks such as running ProvNinja to obtain the filtered feature space gadgets, carefully evaluating the problem space recommendations to reject evasive gadgets from the filtered list of feature space gadgets, and selecting a gadget to implement using different pen-testing frameworks. The result presented in Figure 4.4a, confirms that implementing evasive attacks becomes increasingly time-consuming for larger attack graphs.

Interestingly, we observe that the implementation effort is sublinear in the size of the attack graph as shown in Figure 4.4b. The size of the attack graph is also sublinear in the total graph size because the majority of system events are benign. Compared to the Enterprise APT scenario, the implementation of the Supply Chain APT gadgets requires less time. This is due to the availability of numerous replacement options from the surrogate datasets, which are similar to the defender datasets (as illustrated in §4.8.8). This similarity allows the creation of many gadgets using the surrogate dataset on the defender model.

This can be attributed to two factors. Firstly, the number of rare edges is limited, so generating larger graphs for a specific attack instance does not add any extra rare edges, except for those added due to the attack itself. Secondly, the number of available gadgets is limited, which also limits the ability to generate new paths for larger graphs.

Table 4.8: The detection results of the attacks generated from the benign, surrogate, and random dataset (lower numbers indicate better evasion). Rare edges and the gadget chains are found using the data. The random data is generated by intermixing DARPA TC datasets.

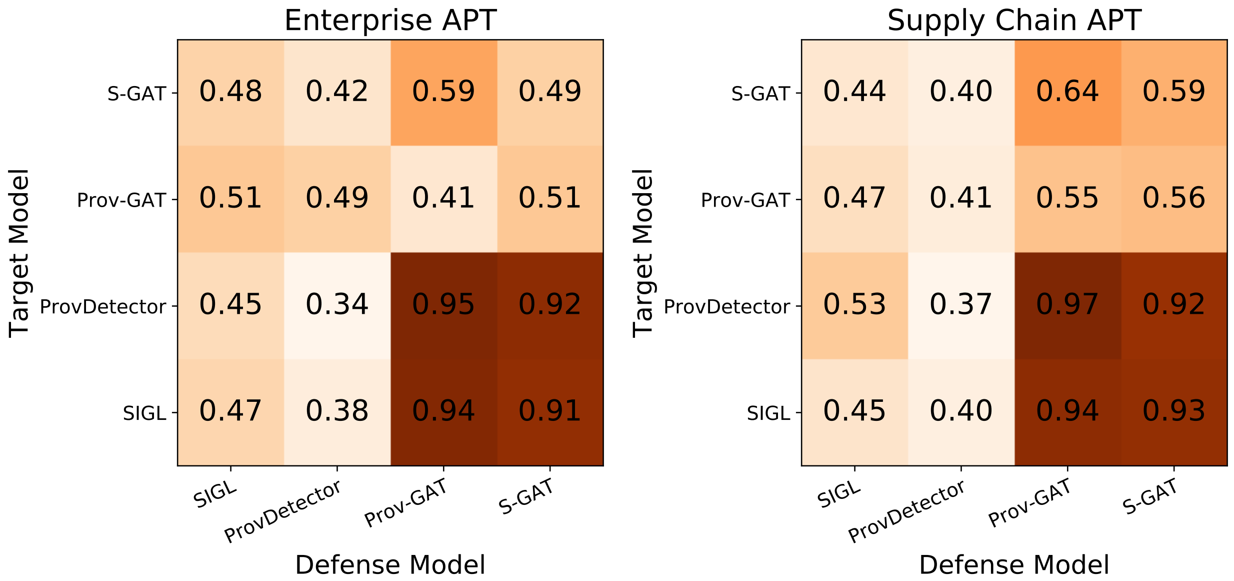
Attack Type	GNN-Based Detectors	Benign Data		Surrogate Data		Random Data	
		Recall	F1	Recall	F1	Recall	F1
Enterprise APT	GAT	0.26	0.35	0.37 *(+.11)*	0.54 *(+.19)*	0.71 *(+.45)*	0.82 *(+.47)*
Supply Chain APT		0.29	0.22	0.44 *(+.15)*	0.53 *(+.31)*	0.96 *(+.67)*	0.91 *(+.69)*
Fileless Malware		0.63	0.72	0.71 *(+.08)*	0.77 *(+.05)*	0.93 *(+.30)*	0.94 *(+.22)*
Enterprise APT	Prov-GAT	0.17	0.28	0.25 *(+.08)*	0.37 *(+.09)*	0.75 *(+.58)*	0.74 *(+.46)*
Supply Chain APT		0.21	0.34	0.28 *(+.07)*	0.56 *(+.22)*	0.88 *(+.67)*	0.92 *(+.58)*
Fileless Malware		0.55	0.66	0.58 *(+.03)*	0.67 *(+.01)*	0.95 *(+.40)*	0.96 *(+.30)*
Average		**0.35**	**0.43**	**0.44 *(+.09)***	**0.57 *(+.15)***	**0.86 *(+.51)***	**0.88 *(+.45)***

Attack Type	Path-based Detectors	Benign Data		Surrogate Data		Random Data	
		Recall	F1	Recall	F1	Recall	F1
Enterprise APT	ProvDetector	0.18	0.15	0.23 *(+.05)*	0.31 *(+.16)*	0.81 *(+.63)*	0.88 *(+.73)*
Supply Chain APT		0.25	0.23	0.35 *(+.10)*	0.30 *(+.07)*	0.94 *(+.69)*	0.93 *(+.70)*
Fileless Malware		0.29	0.41	0.33 *(+.04)*	0.43 *(+.02)*	0.93 *(+.64)*	0.92 *(+.51)*
Enterprise APT	SIGL	0.25	0.36	0.30 *(+.05)*	0.41 *(+.05)*	0.99 *(+.74)*	0.99 *(+.63)*
Supply Chain APT		0.29	0.38	0.38 *(+.09)*	0.43 *(+.05)*	0.95 *(+.66)*	0.92 *(+.54)*
Fileless Malware		0.43	0.51	0.47 *(+.04)*	0.57 *(+.06)*	0.97 *(+.54)*	0.95 *(+.44)*
Average		**0.32**	**0.39**	**0.39 *(+.08)***	**0.50 *(+.11)***	**0.93 *(+.65)***	**0.93 *(+.59)***

4.8.8 Surrogate Dataset Effectiveness

In this section, we evaluate ProvNinja's robustness to surrogate model and its frequency summary with an ablation study and a brief analysis of the domain shift between the DARPA dataset and the benign dataset.

In Table 4.8, we estimate an upper bound on ProvNinja's performance by initially providing it with the true benign dataset (*e.g.,* target network dataset) to create gadgets, which significantly reduces the recall rate and F1 scores of the models. Next, we utilize ProvNinja with the surrogate frequency dataset, incorporating progressively increasing Gaussian noise; we avoid negative event counts by only considering additive noise. Lastly, we try using ProvNinja with fully randomized data, which does not significantly reduce the recall and F1 scores of the models and performs no better than trivial transformations.



(a) Enterprise APT transferability.

(b) Supply Chain APT transferability.

Figure 4.5: Attack transferability visualization. Each cell contains the F1 score of the defense model (columns) that is measured against the evasive attacks crafted against a target model (rows). Lower values (lighter colors) indicate better evasion.

4.8.9 Transferability Evaluation

Attack transferability (Demontis et al., 2019) is an important metric to determine how effective a particular ninja attack is against other models. We evaluated the transferability of the actualized ninja attacks designed for each of the four ML detectors for both our enterprise and supply chain APT scenarios. The key insight is that ninja attacks generated against strong ML detectors (*e.g.,* Prov-GAT and GAT) transfer well to weaker ML detectors (ProvDetector and SIGL). In Figure 4.5, we show that the ninja attacks are highly transferable across model architectures with two notable exceptions: the attacks designed against path-based ML detectors were easily detected by the graph-based ML detectors because ProvNinja-PATH ’s gadgets are not camouflaged; the attacks designed for GAT did not transfer perfectly to Prov-GAT because Prov-GAT sees more information than GAT .

4.9 Related Works

Host- and provenance-based IDS. Various approaches have been proposed (Milajerdi et al., 2019; Hossain et al., 2020, 2018; Fang et al., 2022; Xu et al., 2022) leveraging system provenance to trace stealthy and long-running APT campaigns. Several heuristics have been proposed to prioritize edges (Milajerdi et al., 2019) that are likely to involve malicious semantics referring to a threat intelligence (ATT&CK®, 2022b) source or assigning *tags* (Hossain et al., 2020) that propagate contextual hints to related nodes. Depcomm (Xu et al., 2022) summarizes the graph by creating process-centric communities (clusters) that are connected using system interactions that map the information flow. These communities include important sequences of events that are used for threat detection as they contain important system semantics and are likely to hold the malicious paths.

The large-scale deployment of low-level syscall event collection necessary for system provenance data collection inherently incurs colossal storage pressure, posing substantial challenges in storing and streaming events to support various analysis tasks (Inam et al., 2023). Therefore, substantial research has therefore been dedicated to mitigate the storage pressure while supporting various security analysis tasks(Hossain et al., 2018; Michael et al., 2020). Clearly, utilizing system provenance for cybersecurity is a popular endeavor, but new techniques have not been thoroughly hardened against dedicated adversaries. For defenders and researchers, ProvNinja provides an efficient way to generate attack sequences that are difficult to detect with traditional approaches.

Adversarial ML. Adversarial ML research has gained momentum since Kurakin et al. (Kurakin et al., 2016) first proposed an attack on an established model for image recognition. Since then, over 5,000 adversarial ML research papers have been published in the last decade (Carlini, 2019), including numerous works (Nguyen et al., 2015; Moosavi-Dezfooli et al., 2016) aiming to deceive ML models across different domains and modeling approaches.

Tramer et al. (Tramèr et al., 2017) has proposed adversarial training to increase the robustness of ML models.

Problem space translation. Problem space realization of adversarial examples has been explored by about 80 papers for in various security domains — malicious PDFs, network intrusion detection systems, android malware detection *etc.* Pierazzi et al.(Pierazzi et al., 2020) conducted a comprehensive survey on problem space evasive attacks, providing a framework with four constraints to be considered during the realization. They also implemented their own evasive attacks against an ML-based malware detector analyzing 170K Android malware samples. Using their framework, we discusses unique challenges of problem space realization in the provenance domain, which is empirically evaluated in §4.8.7. Evasive attack realization for the provenance domain has turned out be difficult, as the problem space is distant from the feature space.

Provenance mimicry attacks. Mimicry attacks against provenance-based IDS are advancing and improving rapidly. (Goyal et al., 2023)demonstrated the first versions of such attacks in early 2023, which consistently evaded a wide variety of provenance-based IDS. ProvNinja improves upon the previous work by reducing the number of added system events and extending the tolerable differences between the program distribution in the attacker's dataset and the defender's dataset. In the coming years, as provenance-based IDS gains popularity, we expect to see an arms race of mimicry attacks and defenses, which will ultimately improve the security overall.

Evasive attack transferability is another important research direction in adversarial ML; (Demontis et al., 2019) conducted experiments on linear and non-linear models to evaluate why some adversarial attacks transfer better than others. They provided insights on what aspects contribute to evasive attack transferability and introduced new criteria to measure attack transferability.

CHAPTER 5

EXPLAINING GNN-BASED PIDS – PROVEXPLAINER

5.1 Problem Statement

Our research addresses explainability in GNN-based security models (Cheng et al., 2024; Rehman et al., 2024; Goyal et al., 2024) built on system provenance graphs, tackling a core issue in the security domain. The complexity of explaining GNN decisions is exacerbated by graph structural learning, which adds to the inherent complexity of Neural Networks (NNs). Existing studies on GNN explainability (Yuan et al., 2022; Ying et al., 2019; Yuan et al., 2021; Luo et al., 2020; Herath et al., 2022; Ganz et al., 2023) often fail to effectively map back to system behaviors in the provenance domain. To bridge this gap, our design approach aims to explain GNN decisions by employing a surrogate DT equipped with interpretable security-aware graph structural features.

5.2 Threat Model

Our threat model assumes the integrity of on-device data collection, relying on provenance records secured by existing systems (Wang et al., 2020; Han et al., 2021; Hassan et al., 2019; Mukherjee et al., 2023; Liu, Zhang, Li, Jee, Li, Wu, Rhee, and Mittal, Liu et al.; Cheng et al., 2024; Rehman et al., 2024; Goyal et al., 2024). Our primary objective is to generate security-aware explanations to aid security practitioners and increase their trust in the GNN's decisions. We consider graph-level classification and anomaly detection tasks; explaining GNN decisions in node/edge level tasks is outside the scope of this work. We assume that a knowledgeable security practitioner can differentiate between explanations that align with the ground truth and those that do not, ensuring the validity of our surrogate-based explanations only when they match the GNN model's accurate predictions. Systematically

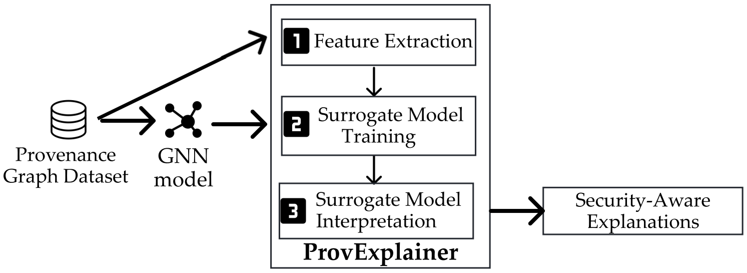

Provenance Graph Dataset
GNN model
1 Feature Extraction
2 Surrogate Model Training
3 Surrogate Model Interpretation
ProvExplainer
Security-Aware Explanations

Figure 5.1: ProvExplainer framework.

generating an accurate and trustworthy ground truth for application, malware, and APT behavior is a challenging open problem. In this dissertation, we approximate the ground truth using publicly available documentation (§2.7). In line with recent literature on GNN explanation (Herath et al., 2022; Warnecke et al., 2019, 2020), adversarial samples are outside the scope of the paper. Creating robust detection and explanation systems that can withstand adversarial manipulation (Goyal et al., 2023; Mukherjee et al., 2023), procedural dataset poisoning, and model manipulate are critical open research problems that are orthogonal to our work.

5.3 ProvExplainer Overview

Given a GNN model built with a system provenance dataset, we apply ProvExplainer in three stages, refer to Figure 5.1.

Stage 1: Extract Security-aware Features (§5.4). Using a data-driven approach (Algorithm 2), we extract security-aware subgraphs (Table 5.1) that exhibit distinctions between benign and anomalous datasets (Figure 5.3). These subgraphs identify attack vectors used by APTs.

Stage 2: Train an Interpretable Surrogate Model (§5.5). Next, we utilize an extensive and diverse system provenance dataset to train an interpretable surrogate DT to agree with the GNN using the extracted features. By decoupling feature engineering from decision-

APT	Attack Vector	Subgraph Structure
Initial Compromise §5.4.1	Staging	Dropper Triangle, Cascade
Establish Foothold §5.4.2	Cloning	Clone Triangle, Probe Triangle
Deepen Access §5.4.3	Inheriting	Kite, Jellyfish
	Sharing	Square
Lateral Movement §5.4.4	Accessing	Exploding Kite, Exploding Square
Look, Learn, and Remain §5.4.5	Exfiltrating	External IP Use

Table 5.1: Summary of program behavior patterns.

making, we *project the GNN's decision boundary onto the interpretable surrogate model's feature space.*

Stage 3: Interpret the Surrogate Model (§5.6). To extract the explanation for a detection using the surrogate DT, we designed Algorithm 3 to extract the graph nodes that contribute to the surrogate DT's decision. These explanations are valid only when the surrogate agrees with the GNN.

ProvExplainer examines graph structural features linked to system actions through extensive data studies supported by security domain expertise. Our security-aware features enabled surrogate DTs to achieve 88% agreement on APT and Fileless Malware detection, and 83% agreement on program classification. We curated an extensive dataset using in-house data collection, APT datasets from various sources (*i.e.,* industry standard DARPA (DARPA, 2019) and ProvNinja (Mukherjee et al., 2023)), and real-world Fileless malware samples from (Barr-Smith et al., 2021) to validate the generalizability of ProvExplainer. ProvExplainer improves precision by 9.14% and recall by 6.97% compared to GNNExplainer (Ying et al., 2019), PGExplainer (Luo et al., 2020), and SubgraphX (Yuan et al., 2021), which are the current state-of-the-art GNN explainers. Furthermore, combining ProvExplainer with state-of-the-art GNN explainers enhances precision and recall by 7.22% and 4.86%, respectively, over the best individual explainer.

Algorithm 2: Graph Structural Feature Extraction

```
Input: Dataset $D$, Node size $n$
Output: Sorted subgraphs by largest difference in count
$subgraphs \leftarrow GetFeasibleSubgraphs(n)$
$count \leftarrow []$
foreach $(graph, label)$ in $D$ do
    $subgraph\_cnt \leftarrow \{sg : CountSubgraphs(graph, sg) \mid sg \in subgraphs\}$
    $counts.append([subgraph\_cnt, label])$
Aggregate the $subgraph\_cnt$ count per $label$
Calculate the $subgraph\_cnt$ difference between the labels
return $subgraphs$ sorted by largest difference in count
```

5.4 Graph Structural Features

Subgraph patterns are the foundation of ProvExplainer's interpretable graph features. They are localized in the provenance graph and correspond to distinct program behaviors in computer systems. The security landscape is continuously evolving, with the MITRE ATT&CK framework documenting over 367 attack vectors (ThreatIntelligence, 2023), and new vectors being added regularly. To adapt to evolving threats (Barbero et al., 2022) by automatically extracting structural features from data, we designed Algorithm 2 to systematically extract subgraph patterns that exhibit a distribution difference between classes within a dataset. We first generate all semantically valid subgraphs of a given size. For example, every edge must contain to at least one process node because only processes can take actions. We then count the instances of each of these subgraphs for each class in the dataset. Finally, we sort the subgraphs by the magnitude of the distribution difference between the classes to obtain the most effective structural features.

However, subgraph pattern mining is resource intensive and scales exponentially in the size of the subgraphs; there are 56 semantically valid 3-node graphlets, and even adding just one more node increases that count to 887. Optimizing the identification and mining of subgraphs is a critical active research area (Paramonov et al., 2019; Jha et al., 2015; Kolda

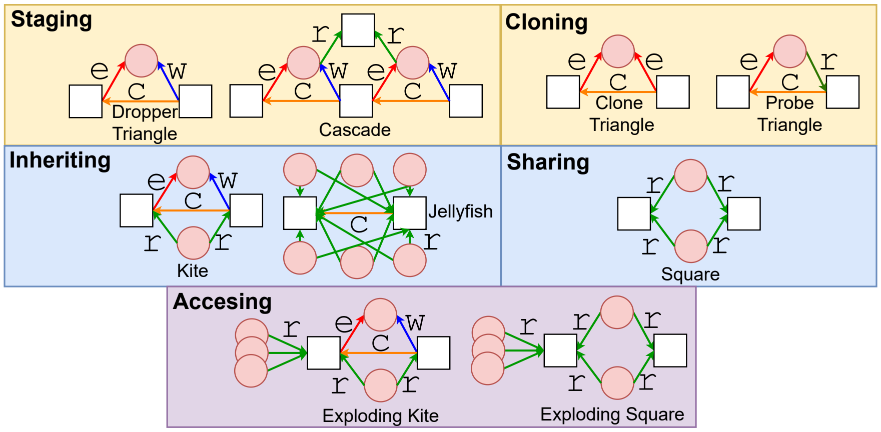


Figure 5.2: Structural Graph Features. Squares are processes and circles are files. *Write* edges are blue, *read* edges are green, *execute* edges are red, and *process creation* edges are orange.

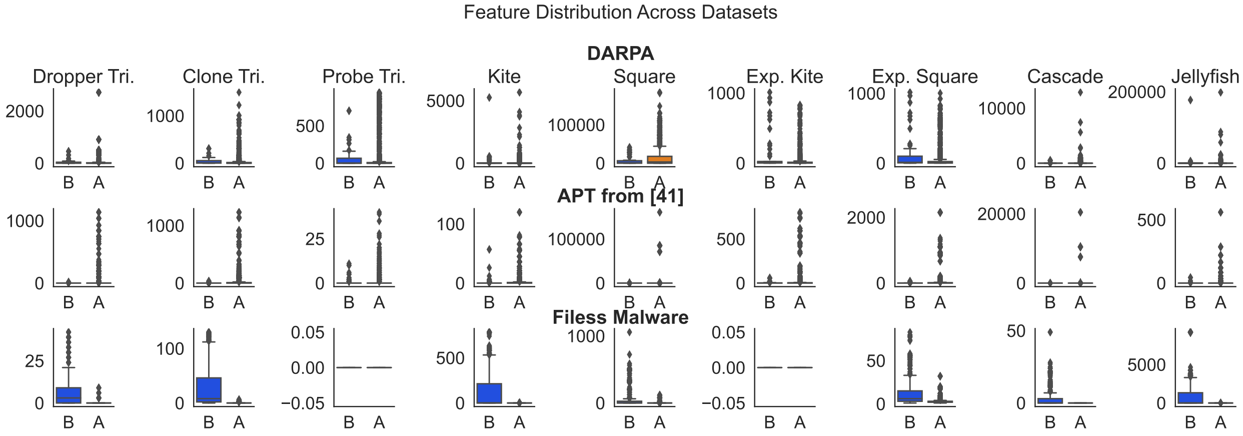


Figure 5.3: Distribution of nine subgraphs (Table 5.1) in different datasets. *B* means benign and *A* means anomaly.

et al., 2014; Seshadhri et al., 2013; Ugander et al., 2013). After analyzing the 56 semantically valid 3-node graphlets and selecting the top 3 strongest contributors, we expedited the analysis of larger graphlets by applying domain expertise and data study to significantly reduce the search space. This way, we obtained 9 patterns that capture common attack vectors (Figure 5.2). We validated these patterns by measuring the distribution difference across data classes (Figure 5.3). Although the *Probe Triangle* and *Exploding Kite* patterns are not well-represented in the Fileless Malware dataset, their strong signals in the APT datasets justify their inclusion. Subgraph structures that map directly to attack vectors aid in generating security-aware explanations.

5.4.1 Initial Compromise

Staging. A prevalent attack vector in conducting the initial compromise (ATT&CK®, 2018b) as seen in DARPA attacks (DARPA, DARPA) is to save the malicious logic in temporary locations (*e.g.,* `\tmp\` or `C:\Users\AppData\Local\Temp`) and then execute it. These temporary locations are often whitelisted by defense mechanisms. The attacker then performs initial compromise by executing the payload. Fileless Malware also contains "dropper" behavior (Sood and Zeadally, 2016; Phillips, 2021). The *Dropper Triangle* and the *Cascade* structures (Figure 5.2) capture the dropper behaviors.

5.4.2 Establishing a Foothold

Cloning. Attackers taking advantage of multiprocessing-based parallelism for redundancy and efficiency to replicate malware instances to overload the system's defenses is emblematic of establishing a foothold (ATT&CK®, 2017). In the *Clone Triangle*, a parent and child process both execute the same program. Clone triangles are also common in benign programs that use multiprocessing as part of their standard workflow, such as `sc.exe` and `explorer.exe`. In the *Probe Triangle*, The attacker first probes the payload and necessary library files by reading them to ensure their existence before executing the payload, which is common in cryptominers (Crowdstrike, 2018). This avoids suspicious events (*e.g.,* accessing nonexistent files) from occurring and triggering defenses. While probing and cloning alone are not sufficient to indicate malicious activity, they amplify the importance of other attack behaviors. The *Clone Triangle* and the *Probe Triangle* (Figure 5.2) efficiently identify the cloning attack vector.

5.4.3 Deepen Access

Inheriting. After gaining initial access and establishing a foothold, attackers scale up their operations to deepen access (ATT&CK®, 2020). These attackers create multiple child processes that execute the same payload. Because the children inherit their objectives from the parent, these parent-child malware pairs read similar library files. Advanced malware writers (Crowdstrike, 2018) also update the configuration inside the malware payload to keep track of the system state to maximize resource consumption (*e.g.,* CPU cycles, RAM, and network bandwidth) without triggering usage alerts. The *Kite* and the *Jellyfish* shape (Figure 5.2) efficiently identify inheriting behavior.

Sharing. Malware such as Banking Trojans (sca, 2019; Grammatikakis et al., 2021) do not directly create malware that access the sensitive documents but rather they create multiple process chains where the last process in the chain does the malicious behavior. APTs also create such process chains to obscure the point of entry and spread throughout the system. In these attacks, processes with distant ancestral relations demonstrate operational similarities by reading the same library files. The *Square* (Figure 5.2) pattern efficiently identifies this resource sharing behavior.

5.4.4 Lateral Movement

Accessing. After deepening access, the attacker reads sensitive resources (*e.g.,* cookies and credential files) to enable lateral movement (ATT&CK®, 2018c) through the system. Capturing the attacker's intermediate objectives reveals additional resources the defender must protect. For example, in one of DARPA Trace's APT scenarios, the attacker reads sensitive configuration and cookies from `/home/admin/.mozilla/firefox/`, and `/usr/local/firefox-xx/obj-x86_64-pc-linux-gnu/` to advance their attack. The *Exploding Kite* and *Exploding Square* shape (Figure 5.2) identifies the resource accessing behavior.

5.4.5 Look, Learn, and Remain

We differentiate network nodes based on whether their IP addresses are internal or external to the system's network. This critical feature requires minimal engineering effort but holds high security importance, as network behavior is extremely hard to hide. DoS malware (Bareckas, 2022) and APT threat actors commonly exfiltrate (ATT&CK®, 2018a) data to their external command and control server. This behavior leads to a network node with a destination IP that is external to the local network.

5.5 Creating Surrogate Decision Trees (DTs) using Graph Structural Features

To obtain explanations from our security-aware graph structural features, we use them to train a global surrogate DT. By training a DT to agree with the predictions of a GNN model, we can interpret the DT to gain insights about the GNN's decision-making process. To achieve the best agreement results, we enhance traditional DT training with data augmentation that iteratively increases the weight of incorrectly classified samples (Jacobs et al., 2022).

We begin with a labelled set of graphs $D_G = (G, Y)$, which is used to train the GNN. GNN's predictions on D_G are collected, yielding $GNN(D_G) = Y'$. To prepare the dataset for training the DT, we extract the graph structural features (§5.4) S and associate them with the GNN's predictions Y' to create a labelled feature dataset $D_F = (S, Y')$, which we split into train, validation, and testing sets to evaluate the surrogate DT.

Leveraging the methods of Jacobs et al. (Jacobs et al., 2022), we use two-layer iterative dataset augmentation to train a series of DT models. At each iteration of the inner loop, all misclassified samples are duplicated to increase their weight in the next iteration; from this series of models, we select the one with the highest agreement among the DTs. This process is repeated several times in the outer loop, then the surrogate model with the highest mean agreement among those high-agreement DTs is selected as the final surrogate model for explanation. This improves the *stability* of the explanations, but at the cost of some agreement on smaller datasets. Because the surrogate model is aggregated over several iterations, the final resulting model is less sensitive to small changes in the training set.

Algorithm 3: Explanation for graph G

```
Input: graph $G$, decision tree $DT$, explanation size $k$, max depth $D$
Output: Top-scoring $k$ nodes from $node\_rankings$
Function ExplainGraph($G$, $DT$, $k$, $D$):
    $node\_rankings \leftarrow \{v : 0 | \forall v \in G.V\}$
    $dp \leftarrow GetDecisionPath(DT, G)$
    for $depth\ d = 1$ to $D$ do
        $shape \leftarrow$ shape corresponding to rule at depth $d$ in $dp$
        $importance \leftarrow \frac{1}{d}$
        foreach node $v \in G.V$ that participates in $shape$ do
            $impact \leftarrow$ # instances of $shape$ that $v$ participates in
            $score \leftarrow importance \cdot impact$
            $node\_rankings[v] \leftarrow \max(node\_rankings[v], score)$
    return the top-scoring $k$ nodes from $node\_rankings$
```

5.6 Interpreting GNN-based IDS Detections Using Surrogate DTs

Once the surrogate DT is trained, not only can we qualitatively analyze the DT for global insights, but we can also use it to explain decisions about individual graphs. Similar to existing explainers, we will use Algorithm 3 to assign an importance score to each node in the graph, then return the most important nodes. Each decision node within the DT consists of a subgraph structure and a threshold. Decision nodes closest to the DT root have the greatest influence on the decision path. In our experiments, we empirically found that considering DT nodes up to a depth of $D = 4$ yielded the best explanations in our datasets; lower depths missed shapes that were necessary for some complex APT scenarios and higher depths incorporated irrelevant system behaviors.

The crux of this methodology lies in ranking the provenance graph nodes based on two pivotal criteria: the importance of the decision node within the decision tree (assigning more importance to nodes that are closer to the root) and the impact of the node on the rules, assessed by the node's biggest contribution to the features used by the rules. Such an approach not only aids in pinpointing critical nodes but also in understanding their roles in the broader context of system interactions. System attributes (*e.g.,* process/file names, and socket IP/port) can be extracted as a post-processing step. Finally, we use the interpretable surrogate DT to construct actionable, security-aware explanations about individual decisions. Because PROVEXPLAINER yields a global surrogate DT, domain experts can analyze it to improve their understanding of the GNN's decision-making process.

Algorithm 4: Combined explanation for graph G

```
Input: graph $G$, explanation size $k$, PROVEXPLAINER's node ranking $R_1$,
       general-purpose explainer's node ranking $R_2$
Output: Combined node ranking
Function EnsembleExplanation($G, k, R_1, R_2$):
    $combined \leftarrow \varnothing$
    for $i = 1$ to $k$ do
        if $i \mod 2 = 1$ then
            $combined \leftarrow combined \cup argmax_{v \in G.V \setminus combined}(R_1(v) \mid v \notin combined)$
        else
            $combined \leftarrow combined \cup argmax_{v \in G.V \setminus combined}(R_2(v) \mid v \notin combined)$
    return $combined$
```

5.7 Combining SOTA GNN Explanation Methods with ProvExplainer

When explainers are viewing the GNN's decisions from different angles, it is often beneficial to consider input from multiple explanations to create a combined view of the important elements of the graph. In Algorithm 4, we present a method for merging the top ranking nodes from multiple explainers' perspectives. By going through the explainers in a roundrobin fashion, we ensure that the top-ranking nodes from each explainer are fairly represented in the final result.

ProvExplainer provides explanations that are guided towards security-relevant graph structures, while traditional GNN explainers focus entirely on structures that are important to the GNN model.

5.8 Evaluation

In this section, we evaluate ProvExplainer's effectiveness in explaining stealthy attacks. We aim to answer the following research questions (RQs):

- **RQ1: Explanation Accuracy.** Can ProvExplainer explain APT and Fileless malware detection (§5.8.5, and §5.8.6)?
- **RQ2: Comparison with SOTA GNN Explainers.** How do the explanations of ProvExplainer compare against those of SOTA GNN explainers (GNNExplainer (Ying et al., 2019), PGExplainer (Luo et al., 2020), and SubgraphX (Yuan et al., 2021)) (§5.8.7)?
- **RQ3: Explanation Ensemble.** Can ProvExplainer's explanations be combined with those of SOTA GNN explainers to improve explanation stability (§5.8.7)?

5.8.1 Evaluation Protocols

For APT detection, Fileless Malware detection, and program classification tasks, we leveraged three kinds of datasets: *(1)* publicly available APT attack simulations (DARPA, 2019; Mukherjee et al., 2023), *(2)* execution traces of Fileless Malware (Barr-Smith et al., 2021), and *(3)* program execution traces collected from our in-house testbed. We implemented two general purpose SOTA GNN models: GAT (Veličković et al., 2017) and GraphSAGE (Hamilton et al., 2017), following the approach of recent explanation literature (Herath et al., 2022; Kosan et al., 2023). Recent GNN based anomaly detection systems (Zengy et al., 2022; Rehman et al., 2024) rely on custom node and edge embeddings for security tasks, so we did not evaluate against these specialized solutions. We conducted an ablation study and evaluated the explanations given by ProvExplainer against those of SOTA GNN explainers (Ying et al., 2019; Luo et al., 2020; Yuan et al., 2021).

Program Classification. We choose popular programs whose runtime behaviors are dictated by command line arguments, such as `python` and `powershell.exe`. We also selected `firefox.exe` because it is a versatile program with different usages and almost all the FiveDirections APT attacks in the DARPA dataset exploited `firefox.exe` to gain access to the system. Therefore, we wanted to explore how the command line arguments relate to the behavior of `firefox.exe`. The full list of categories for each program is provided in Table 5.2, and representative graphs are illustrated in Figure 5.4.

APT Attack Detection. The GNN are tasked with detecting stealthy APT scenarios from the publicly available DARPA Transparent Computing (TC) Data Release (DARPA, 2019) and ProvNinja (Mukherjee et al., 2023). The DARPA dataset comprises fine-grained event collection implemented across diverse OSes, providing a solid foundation for advanced security research. We selected three of the most prevalent datasets commonly used in other security papers (Zengy et al., 2022; Cheng et al., 2024; Rehman et al., 2024; Goyal et al., 2024): FiveDirections, Trace, and Theia. The APTs were designed to attack a system which consists of long-running processes and captured the stealthy attack vectors frequently employed by advanced adversaries. However, these tasks were conducted in a simulated environment and lasted for two weeks, involving a limited number of hosts. Therefore, we also evaluated against the APT scenarios conducted by ProvNinja, which was composed of hosts under realistic workloads.

Fileless Malware Detection. We target a family of stealthy malware samples based on previous research (Barr-Smith et al., 2021) that impersonate well-known benign programs.

The Fileless malware uses the benign program binary to inject malicious payload into a process, and can trivially evade conventional security solutions. But their behavior is captured in system provenance graphs that are utilized by GNN based detectors to effectively detect them. Following the guidelines of previous research (Küchler et al., 2021; Avllazagaj et al., 2021) that warned about experimental bias and freshness, we carefully chose the sample to ensure that they are representative of the malwares found in the wild and are active. We chose noteworthy instances for case studies in §5.9.

Evaluation Metrics. In our evaluation of ProvExplainer, we focus on two critical aspects. The first is the agreement of the surrogate DTs with the GNN model, which we measure using the weighted macro averaged (WMA) F1 score of the surrogate DT's predictions with respect to the GNN's predictions. The *agreement* metric gauges the faithfulness of the DT in replicating the conclusions of GNNs. The choice of the WMA F1 score accounts for the data imbalance issue prevalent in the anomaly detection datasets.

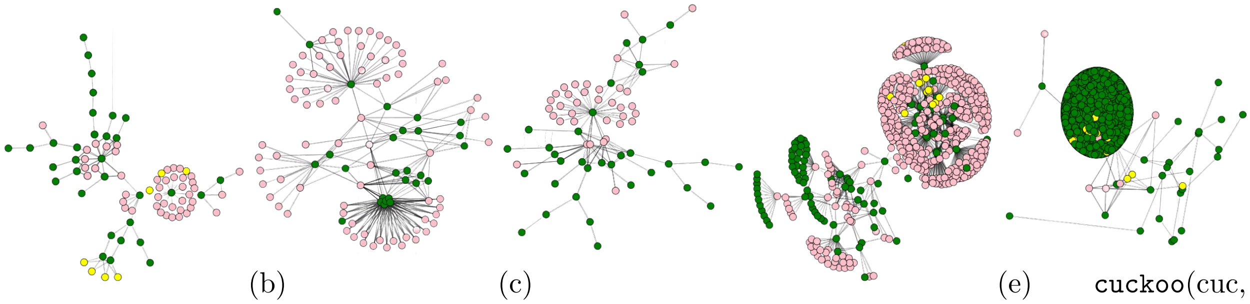
To evaluate ProvExplainer's and SOTA GNN explainers' proficiency in identifying security-relevant entities, we define precision and recall metrics with respect to documented entities §2.7. A graph explanation method EM will yield a total ordering over V for a given graph $G = (V, E)$ and graph model M. Let V_k be the top k nodes according to $EM(G, M)$. Let D be the set of documented entities. *Precision* is the proportion of explanation nodes that are documented, and *recall* is the fraction of documented entities retrieved: $precision(V_k, k, D) = \frac{|V_k \cap D|}{k}$, and $recall(V_k, D) = \frac{|V_k \cap D|}{|D|}$.

5.8.2 Evaluation Tasks

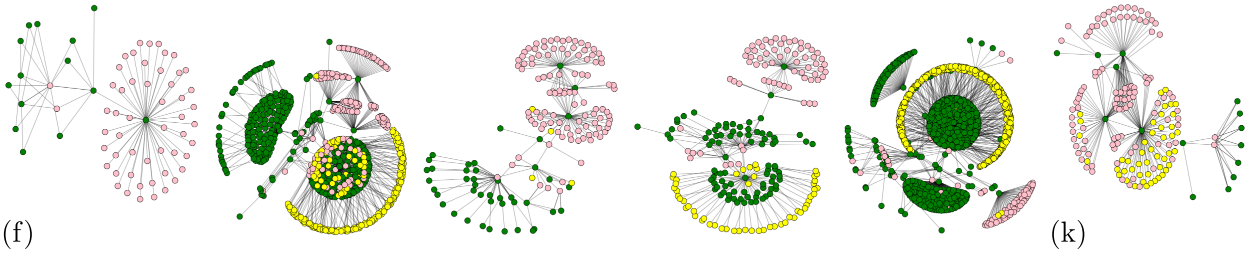
APT Detection. We evaluated using two different datasets: the DARPA Transparent Computing (TC) Data Releases (DARPA, 2019) and APT attack detection dataset from ProvNinja (Mukherjee et al., 2023). This dataset, encompassing various OSes, provides a comprehensive basis for advanced security research. The DARPA APTs were designed to attack a system which consists of long-running processes and captured the stealthy attack vectors frequently employed by advanced adversaries. We particularly focused on three DARPA datasets used by previous studies (Zengy et al., 2022; Cheng et al., 2024; Rehman et al., 2024; Goyal et al., 2024): FiveDirections, Trace, and Theia. However, these tasks were conducted in a simulated environment and lasted for two weeks, involving a limited number of hosts. Therefore, we also evaluated against the APT scenarios conducted during ProvNinja (Mukherjee et al., 2023), which were performed with realistic benign background workloads.

Fileless Malware Detection. For Fileless Malware detection, we targeted a family of stealthy malware samples that impersonate benign programs, evading conventional security solutions but are detectable by GNN-based provenance analysis. The malware samples were chosen in accordance with guidelines from the literature (Küchler et al., 2021; Avllazagaj et al., 2021) to minimize experimental bias and ensure freshness. The Fileless Malware dataset includes various categories (Küchler et al., 2021), including banking Trojans, ransomware, spyware, and malware installers, with detailed statistics presented in Table 5.3 in the appendix.

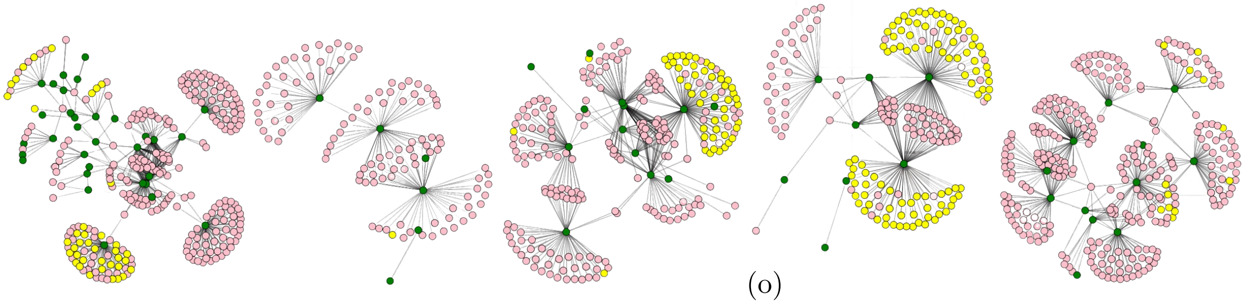
Program Classification. Our program classification task asks the GNN to distinguish between operational modes of versatile system programs whose runtime behaviors influenced by command line arguments. For example, we want to determine which `python` program is running between `certbot`, `update-apt-xapian-index`, `unattended-upgrade`, `decompyle3`, and `cuckoo`. We have similar datasets for `powershell.exe` and `firefox.exe`, which are described in detail in the appendix in Table 5.2. Our data collection, approved by the Institutional Review Board (IRB), involved system event data from volunteers' Windows and Linux hosts, totaling 14 TB over 13 months. This data encompassed a variety of user workloads including students, researchers, developers, and administrators. The program classification task provides a reference point for this study because the classes are balanced, stabilizing the GNN's decisions.



(a) `certbot` (b) `update-apt-xapian-index` (c) `unattended-upgrade` (d) `decompyle3` (e) `cuckoo`(cuc, 2019)



(f) `unrestricted` (g) `allSigned` (h) `restricted` (i) `bypass` (j) `noLogo` (k) `noInputFormat`



(l) `moz_log` (m) `osint` (n) `jsInit` (o) `win32LockedDown` (p) `file`

Figure 5.4: Representative graphs of the different categories for `python` (linux), `powershell.exe` and `firefox.exe`.

5.8.3 Evaluation Datasets

With the approval and oversight of our university's Institutional Review Board (IRB), we solicited written informed consent from volunteers to participate in a long-running provenance data collection project. Using Linux kernel audit(Redhat, 2017) and Windows ETW (Microsoft, 2015) event tracing framework, we collected system provenance data. Our volunteers performed a variety of workloads as students, researchers, developers, and administrators. In aggregate, our volunteers have helped us collect system event data from 9 Windows hosts and 12 Linux hosts over 13 months, yielding 14TB of system event data.

Classification Programs. From Linux, we choose distinctive `python`, `powershell.exe` and `firefox.exe` behavior for program classification as shown in Figure 5.4. Besides the fact that all of these behaviors stem from the binary running different scripts at runtime, these programs are one of the most versatile program. Programs such as `python` and `powershell.exe` are two of the most popular scripting languages. Due to their convenient syntax, widespread availability, and mature ecosystem, many malware authors favor `python` and `powershell.exe` for prototyping malware and attack payloads. The developer community has gradually adopted python for important tasks such as system utilities, language decompilers, certification management, malware analysis, and *etc.* A detailed list of different behaviors that we choose for the evaluation are listed in Table 5.2 and the dataset statistics presented in Table 5.4.

DARPA APT Dataset. Each DARPA TC engagement is divided to multiple subsets, each corresponding to the organization which collected the data. The THEIA subset was collected on Ubuntu 12.04 using the BEEP framework, which partitions long-running programs to reduce the effect of dependency explosion, resulting in a very granular dataset. TRACE was collected on Ubuntu 14.04 and FiveDirections on Windows 10 hosts. The datasets consist of various attacks (§5.9), *e.g.,* exploiting firefox backdoor, exploiting malicious firefox extension, phishing user credentials using attachment with malicious macro. The benign activity was created by running custom scripts that simulate benign user activity. The benign behavior does not extend beyond what is defined in the scripts, which leads to a low variety of programs present. Detailed dataset statistics found in Table 5.3.

Fileless malware. The Fileless malware dataset used for detection task contained malware from four major categories: banking Trojans, ransomware, spyware, and malware installer. Blackmoon (Küchler et al., 2021) is a banking Trojan that steals banking credentials from victim machines and spreads through spam and compromised download links. To camouflage their interactions with `.dlls` and temporary files, Blackmoon masquerades as `svchost.exe`.

Ulise (Küchler et al., 2021) is spyware that is a multi-purpose Trojan that can establish remote access connections, capture keyboard input, collect system information, download-/upload files, drop other malware into the infected system, and perform encryption. Because it propagates through the network sockets and interacts with many system files, it masquerades as `python.exe` as it needs a target that exhibits diverse behavioral pattern. These Trojans also masqueraded as `explorer.exe`, `wmic.exe`, `reg.exe`, `sc.exe`. Detailed dataset statistics found in Table 5.3.

APT Dataset from (Mukherjee et al., 2023). Two different APT attacks were described in the recent literature (Mukherjee et al., 2023), Enterprise APT and Supply-Chain APT. In the Enterprise scenario, the adversary sends a phishing email for the initial penetration to take over the network-wide domain controller. Eventually, the attacker launches a large-scale database leakage attack. In the Supply-Chain scenario, the attacker gains *initial access* by committing seemingly benign patches to a public repository, which are later pulled by the victim and included in an organization's `Docker` image. The attacker then performs arbitrary read and write from the docker container instances to *exfiltrate* the organization's data. Interested readers can refer to (Mukherjee et al., 2023) for more details and Table 5.3 for dataset statistics.

5.8.4 Dataset Statistics

Anomaly Detection Dataset. The dataset statistics for the anomaly detection dataset is seen in Table 5.3. The Fileless Malware samples were downloaded from (VirusTotal, 2021) and selected from recent studies, (Grammatikakis et al., 2021; Küchler et al., 2021; Avllazagaj et al., 2021; Barr-Smith et al., 2021). The benign provenance graphs for the anomaly detection dataset were sourced from the DARPA dataset, which contained 5,695 benign graphs with an average of 669.81 vertices and 982.18 edges (Table 5.3); the Fileless Malware dataset, which contained 19,422 benign graphs with an average of 60.44 vertices and

Table 5.2: Datasets used for classification: `python`(linux), `powershell.exe` and, `firefox.exe`.

Program	Categories	Description
		Linux
`python`	`cerbot`	Program is a ssl certificate management/update utility program.
	`update-apt-xapian-index`	System programs that rebuild the indexes index to sorts of extra information, such as Debtags tags, and package ratings.
	`unattended-upgrade`	System program that updates the linux distribution without user interface
	`decompyle3`	Custom python decompilation program that recovers the source code of python programs from their compiled bytecode instructions.
	`cuckoo`	Sandbox utility that lets the user run programs inside a sandbox and capture the system level interactions.
		Windows
`powershell.exe`	`unrestricted`	Executes a unrestricted hidden script to disable unused SMBv1 protocol features.
	`allSigned`	Sets output encoding to UTF-8 and starts opening a file stream for only signed scripts.
	`restricted`	Executes a simple display command under a restricted policy, allowing no scripts.
	`bypass`	Bypasses the execution policy in PowerShell.
	`noLogo`	Runs PowerShell bypassing the execution policy, without the PowerShell logo, user profile, and interactively.
	`noInputFormat`	Runs PowerShell in a non-interactive mode, with input format set to none and output format set to text.
`firefox.exe`	`moz_log`	Enables detailed logging and runs a background tasks (*e.g.,* updates).
	`osint`	Opens a specific URL, invoked by an external application (*e.g.,* `discord`).
	`jsInit`	Starts a internal content rendering process with detailed parameters for communication and preferences.
	`win32LockedDown`	Initializes a secured sandbox environment for content in Firefox with specific build details.
	`file`	Opens a local HTML or PDF file.

Table 5.3: APT and Fileless Malware graph statistics.

Applications	# of Benign Graphs	# of Anomaly Graphs	Avg # of Benign Nodes / Edges	Avg # of Anomaly Nodes / Edges
		DARPA APT Dataset (DARPA, 2019)		
`Trace`	1883	8	735.35 / 957.56	836.15 / 946.75
`Theia`	2858	9	559.47 / 979.59	913.91 / 987.31
`FiveDirections`	954	13	906.22 / 971.91	959.43 / 973.63
Average	**1898.33**	**10.00**	**669.81 / 982.18**	**777.08 / 1011.19**
		APT Dataset from (Mukherjee et al., 2023)		
`Enterprise`	3079	1836	90.22 / 85.13	73.73 / 76.88
`Supply-Chain`	3212	1092	65.02 / 40.77	61.09 / 54.33
Average	**3145.50**	**1464.00**	**55.94 / 83.29**	**58.38 / 51.69**
		Fileless Malware from (Barr-Smith et al., 2021)		
`explorer.exe`	399	40	66.18 / 59.94	44.10 / 56.45
`wmic.exe`	876	11	88.56 / 81.58	87.36 / 101.54
`reg.exe`	309	116	60.18 / 52.93	78.91 / 131.87
`sc.exe`	621	7	44.08 / 37.49	38.42 / 52.28
`python.exe`	15585	426	89.95 / 83.15	57.98 / 79.33
`svchost.exe`	1632	443	52.42 / 46.42	51.97 / 81.31
Average	**2667.00**	**90.84**	**60.44 / 54.15**	**63.72 / 89.93**

54.15 edges; and the APT dataset, which contained 6,291 benign graphs with an average of 55.94 vertices and 83.29 edges. The corresponding anomaly graphs are sourced from the same datasets: The DARPA dataset contained 30 anomalous graphs with an average of 777.08

Table 5.4: Classification dataset graph statistics.

Program	Categories	Avg Nodes / Edges	Graphs
	Linux		
`python`	`certbot`	74.85/ 149.94	4955
	`update-apt-xapian-index`	169.78/ 302.37	4922
	`unattended-upgrade`	742.24/ 312.56	4966
	`decompyle3`	90.67/ 169.61	5010
	`cuckoo`	305.26/ 403.99	5072
Average		**190.62 / 353.63**	**24925.00**
	Windows		
`powershell.exe`	`unrestricted`	71.20 / 172.10	2704
	`allsigned`	815.50 / 917.10	1123
	`restricted`	100.00 / 129.60	1002
	`bypass`	639.40 / 713.40	1074
	`noLogo`	691.40 / 743.70	780
	`noIputFormat`	195.00 /283.70	614
`firefox.exe`	`moz_log`	249.80 / 340.40	918
	`osint`	95.50 / 115.60	323
	`jsInit`	144.50 / 183.10	14051
	`win32LockedDown`	102.30 / 124.20	741
	`file`	143.80 / 222.80	91
Average		**295.31 / 358.70**	**11710.50**

vertices and 1011.19 edges (Table 5.3); the Fileless Malware contained 1,043 anomalous graphs with an average of 63.72 vertices and 89.93 edges; and the APT dataset contained 2,928 anomalous graphs with an average of 58.38 vertices and 51.69 edges.

Classification Dataset. For program classification, we chose `python` from the Linux environment and `powershell.exe` and `firefox.exe` from the Windows environment for our classification evaluation. `Python` and `powershell.exe` are language interpreters with flexible behaviors. In the DARPA datasets, attackers leveraged `firefox.exe` in many scenarios to gain initial access. Therefore, we wanted to investigate how does `firefox.exe` behave under different benign workloads. The provenance graphs from Linux environment contained 190.62 vertices, 353.63 edges and 24925 graphs and Windows environment contained 295.31 vertices, 358.70 edges and 11710 graphs, on average (Table 5.4). From the program description in Table 5.2 it is clear that each type of program has distinctive behavior.

We chose five different classes of python programs as shown in Table 5.2. `certbot` is a ssl certificate management utility program, `apt-xapian-index` and `unattended-upgrade` are system programs that rebuild the indexes and updates the linux distribution without user interface, respectively. `cuckoo` (cuc, 2019) is a sandbox utility that runs programs inside a sandbox and capture the system level interactions. `certbot` and `cuckoo` will have several outgoing network connections, but `certbot`'s outbound network connection will have higher number of connection to external IPs as compared to `cuckoo`s. `apt-xapian-index` and `unattended-upgrade`, even though they are system utilities, behave differently in the sense that `unattended-upgrade` reads from many shared files such as `.lock` files and writes to specific `.log` files, but `apt-xapian-index` reads a lot of shared `.so` and `.db` files; `apt-xapian-index` also creates children who read from the same file `.db` files, so there is information transfer.

`Powershell.exe` classes include four different kinds of execution policy, `unrestricted`, `allsigned`, `restricted`, `bypass`, and two kinds of configuration `noLogo`, and `noInputformat`. The class `unrestricted` permits the execution of all PowerShell scripts, including those downloaded from the internet *i.e.,* for application updates. The `allsigned` class enhances security by requiring that all scripts and configuration files be signed by a trusted publisher before execution. The default policy, `restricted`, offers a high level of security by preventing the execution of any scripts. In contrast, `bypass` allows scripts to run without any restrictions, which is commonly employed for troubleshooting purposes. The `noLogo` class is used to start PowerShell without displaying the logo and to disable prompts, scripts, and interactive input, making it suitable for automation tasks. Finally, `noInputformat` specifies that PowerShell no input is expected, a mode to execute scripts in a non-interactive environment without any external input.

The classes for `firefox.exe` encompass: `moz_log`, `osint`, `jsInit`, `win32kLockedDown`, and `file`. These classes reflect various Firefox execution modes and debugging options. The

Table 5.5: Surrogate DTs have high agreement with the decisions of GNN measured using the WMA F1 score. Grey cells low agreement with the GNN (discussed in §5.8.5).

Dataset	GAT	ProvDT (agree w/ GAT)	GraphSAGE	ProvDT (agree w/ GraphSAGE)
APT Dataset from (Mukherjee et al., 2023)				
Enterprise	0.97	0.93	0.94	0.92
Supply-Chain	0.87	0.90	0.84	0.91
Average	0.91	0.92	0.89	0.92
DARPA APT Dataset (DARPA, 2019)				
FiveDirections	0.72	0.88	0.69	0.87
Trace	0.67	0.90	0.65	0.92
Theia	0.69	0.90	0.59	0.90
Average	0.69	0.89	0.64	0.90
Fileless Malware from (Barr-Smith et al., 2021)				
`explorer.exe`	0.99	0.79	0.95	0.67
`wmic.exe`	0.89	0.66	0.87	0.89
`reg.exe`	0.94	0.96	0.90	0.93
`sc.exe`	0.99	0.99	0.93	0.63
`python.exe`	0.97	0.79	0.95	0.75
`svchost.exe`	0.99	0.84	0.99	0.81
Average	0.96	0.83	0.88	0.83
Program Classification				
`python`	0.90	0.85	0.77	0.71
`powershell.exe`	0.92	0.97	0.87	0.97
`firefox.exe`	0.83	0.80	0.65	0.70
Average	0.88	0.87	0.76	0.79

`moz_log` class is used to enable detailed logging in Firefox. The `osint` class indicates that Firefox is being launched for an external URL or file, typically seen in scenarios where the browser is invoked by other applications or operating system components (*e.g.,* `discord.exe` or `slack.exe`). The `jsInit` is a technical flag related to JavaScript engine initialization for internal debugging. The `win32kLockedDown` class is associated with security features, where Firefox operates with restricted access to certain system calls, typically used to mitigate risks of kernel vulnerabilities. The `file` class captures the direct opening of local files.

5.8.5 Graph Structural Feature Evaluation

To answer **RQ1**, Table 5.5 demonstrates the effectiveness of surrogate DTs in mirroring the decision process of GNN models like GAT and GraphSAGE. Agreement between the surrogate DT and the GNN is an important metric for two reasons: *(1)* surrogate explanations are only valid when the surrogate agrees with the GNN, and *(2)* high agreement indicates that the surrogate model is a good approximation of the GNN's decision-making process.

The APT datasets exhibit high agreement (> 87%) with the GNN models, but the Fileless Malware and classification datasets only showed moderate agreement (> 79%). The graph structural features were designed with APT structures in mind, and are optimized to capture malicious behavior. The surrogate DTs' superior performance in APT datasets can be attributed to ProvExplainer's optimized graph shapes that effectively capture the behaviors of the APT stages. Focusing on the `python` and `firefox.exe` classification datasets, where GraphSAGE performed relatively poorly, the surrogate DTs were unable to closely approximate the GraphSAGE model using the graph structural features. For example, `firefox.exe` creates *Jellyfish* shapes both when users browse for content and when system programs use it to download updates, since in both instances `firefox.exe` creates a child that reads the same system files (*e.g.,* `System32`). The ablation study, Table 5.6 shows that the *Jellyfish* shape performs the worst for `firefox.exe`.

In the Fileless Malware datasets, two distinct scenarios emerge. First, in `explorer.exe` and `python.exe`, the surrogate DTs show poor agreement with both the GAT and GraphSAGE models. These datasets are dominated by malware using stealthy techniques such as *living-off-the-land*, which involve memory object interactions which are currently not captured in our provenance graphs. The absence of the distinguishing features impairs the effectiveness of ProvExplainer. Secondly, there are cases where the surrogate DT's agreement is poor with only GAT (`wmic.exe`) or only GraphSAGE (`sc.exe`). This variation arises because ProvExplainer's data augmentation methods aim to enhance stability at the expense of agreement. Consequently, in datasets with very few anomalous examples, like those of `wmic.exe` and `sc.exe` (Table 5.3), the surrogate DTs' performance becomes unstable which is an inherent limitation of the surrogate model approach (Jacobs et al., 2022).

Table 5.6: Agreement of surrogate DTs with the GAT model across different feature subsets. The best feature subsets are highlighted.

Dataset	Number of nodes and edges	Security Domain Features §5.4					**All** Security Domain Features
		Triangles	Squares and Kites	Exploding Shapes	Cascade and Jellyfish	Internal vs External IPs	
DARPA APT Dataset (DARPA, 2019)							
`FiveDirections`	0.58 *(-0.30)*	0.51 *(-0.37)*	0.85 *(-0.03)*	0.85 *(-0.03)*	0.84 *(-0.04)*	0.79 *(-0.09)*	0.88
`Trace`	0.61 *(-0.29)*	0.60 *(-0.30)*	0.84 *(-0.06)*	0.84 *(-0.06)*	0.88 *(-0.02)*	0.79 *(-0.11)*	0.90
`Theia`	0.68 *(-0.22)*	0.75 *(-0.15)*	0.86 *(-0.04)*	0.84 *(-0.06)*	0.88 *(-0.02)*	0.82 *(-0.08)*	0.90
Average	**0.62 *(-0.27)***	**0.62 *(-0.27)***	**0.85 *(-0.04)***	**0.84 *(-0.05)***	**0.87 *(-0.03)***	**0.80 *(-0.09)***	**0.89**
APT Dataset from (Mukherjee et al., 2023)							
`Enterprise`	0.72 *(-0.21)*	0.78 *(-0.15)*	0.83 *(-0.10)*	0.91 *(-0.02)*	0.66 *(-0.27)*	0.88 *(-0.05)*	0.93
`Supply-Chain`	0.75 *(-0.15)*	0.70 *(-0.20)*	0.87 *(-0.03)*	0.83 *(-0.07)*	0.63 *(-0.27)*	0.67 *(-0.23)*	0.90
Average	**0.73 *(-0.18)***	**0.74 *(-0.18)***	**0.85 *(-0.07)***	**0.87 *(-0.05)***	**0.65 *(-0.27)***	**0.78 *(-0.14)***	**0.92**
Fileless Malware from (Barr-Smith et al., 2021)							
`explorer.exe`	0.57 *(-0.22)*	0.76 *(-0.03)*	0.77 *(-0.02)*	0.77 *(-0.02)*	0.46 *(-0.33)*	0.69 *(-0.10)*	0.79
`wmic.exe`	0.49 *(-0.17)*	0.51 *(-0.15)*	0.55 *(-0.11)*	0.57 *(-0.09)*	0.62 *(-0.04)*	0.55 *(-0.11)*	0.66
`reg.exe`	0.87 *(-0.09)*	0.88 *(-0.08)*	0.91 *(-0.05)*	0.85 *(-0.11)*	0.88 *(-0.08)*	0.92 *(-0.04)*	0.96
`sc.exe`	0.49 *(-0.50)*	0.80 *(-0.19)*	0.97 *(-0.02)*	0.94 *(-0.05)*	0.80 *(-0.19)*	0.84 *(-0.15)*	0.99
`python.exe`	0.71 *(-0.08)*	0.78 *(-0.01)*	0.79 *(0.00)*	0.77 *(-0.02)*	0.74 *(-0.05)*	0.76 *(-0.03)*	0.79
`svchost.exe`	0.74 *(-0.10)*	0.84 *(0.00)*	0.83 *(-0.01)*	0.83 *(-0.01)*	0.82 *(-0.02)*	0.81 *(-0.03)*	0.84
Average	**0.65 *(-0.19)***	**0.76 *(-0.08)***	**0.80 *(-0.04)***	**0.79 *(-0.05)***	**0.72 *(-0.12)***	**0.76 *(-0.08)***	**0.84**
Program Classification							
`python`	0.53 *(-0.32)*	0.77 *(-0.08)*	0.83 *(-0.02)*	0.75 *(-0.10)*	0.79 *(-0.06)*	0.82 *(-0.03)*	0.85
`powershell.exe`	0.63 *(-0.34)*	0.81 *(-0.16)*	0.75 *(-0.22)*	0.82 *(-0.15)*	0.65 *(-0.32)*	0.95 *(-0.02)*	0.97
`firefox.exe`	0.35 *(-0.45)*	0.38 *(-0.42)*	0.41 *(-0.39)*	0.44 *(-0.36)*	0.46 *(-0.34)*	0.61 *(-0.19)*	0.80
Average	**0.53 *(-0.34)***	**0.77 *(-0.10)***	**0.83 *(-0.04)***	**0.79 *(-0.08)***	**0.82 *(-0.05)***	**0.75 *(-0.12)***	**0.87**

ProvExplainer's graph structural features enable surrogate DTs to approximate GNN models' decision-making process on in-distribution data. Although the features are sensitive to the data distribution, additional features can be extracted to extend support to new distributions.

5.8.6 Ablation Study

Table 5.6 shows the contributions of each structural feature to the overall agreement of surrogate DTs approximating a GAT model. We take the number of nodes and edges in the graph as a simple baseline, which obtains an average F1 score of 0.63 across our datasets, demonstrating that graph size alone is insufficient. Attack subgraphs typically represent

Table 5.7: WMA F1 of surrogate DTs approximating a GraphSAGE model across different feature subsets. The best feature subsets are bolded.

Dataset	Number of nodes and edges	Security Domain Features §5.4					**All** Security Domain Features
		Triangles	Squares and Kites	Exploding Shapes	Cascade and Jellyfish	Internal vs External IPs	
DARPA APT Dataset (DARPA, 2019)							
`FiveDirections`	0.58 *(-0.32)*	0.74 *(-0.16)*	0.88 *(-0.02)*	0.88 *(-0.02)*	0.71 *(-0.19)*	0.75 *(-0.15)*	0.90
`Trace`	0.61 *(-0.31)*	0.73 *(-0.19)*	0.81 *(-0.11)*	0.86 *(-0.06)*	0.86 *(-0.06)*	0.81 *(-0.11)*	0.92
`Theia`	0.58 *(-0.32)*	0.79 *(-0.11)*	0.81 *(-0.09)*	0.87 *(-0.03)*	0.84 *(-0.06)*	0.82 *(-0.08)*	0.90
Average	**0.59 *(-0.32)***	**0.75 *(-0.15)***	**0.83 *(-0.07)***	**0.87 *(-0.04)***	**0.80 *(-0.10)***	**0.79 *(-0.11)***	**0.91**
APT Dataset from (Mukherjee et al., 2023)							
`Enterprise`	0.72 *(-0.18)*	0.78 *(-0.12)*	0.70 *(-0.20)*	0.63 *(-0.27)*	0.63 *(-0.27)*	0.67 *(-0.23)*	0.90
`Supply-Chain`	0.75 *(-0.16)*	0.76 *(-0.15)*	0.87 *(-0.04)*	0.87 *(-0.04)*	0.84 *(-0.07)*	0.87 *(-0.04)*	0.91
Average	**0.73 *(-0.17)***	**0.77 *(-0.14)***	**0.78 *(-0.12)***	**0.75 *(-0.16)***	**0.73 *(-0.17)***	**0.77 *(-0.14)***	**0.91**
Fileless Malware from (Barr-Smith et al., 2021)							
`explorer.exe`	0.57 *(-0.10)*	0.66 *(-0.01)*	0.67 *(0.00)*	0.65 *(-0.02)*	0.63 *(-0.04)*	0.54 *(-0.13)*	0.67
`wmic.exe`	0.49 *(-0.40)*	0.85 *(-0.04)*	0.88 *(-0.01)*	0.81 *(-0.08)*	0.82 *(-0.07)*	0.87 *(-0.02)*	0.89
`reg.exe`	0.87 *(-0.06)*	0.37 *(-0.56)*	0.93 *(0.00)*	0.88 *(-0.05)*	0.92 *(-0.01)*	0.49 *(-0.44)*	0.93
`sc.exe`	0.49 *(-0.44)*	0.84 *(-0.09)*	0.91 *(-0.02)*	0.87 *(-0.06)*	0.91 *(-0.02)*	0.83 *(-0.10)*	0.93
`python.exe`	0.71 *(-0.04)*	0.68 *(-0.07)*	0.74 *(-0.01)*	0.74 *(-0.01)*	0.72 *(-0.03)*	0.67 *(-0.08)*	0.75
`svchost.exe`	0.74 *(-0.07)*	0.81 *(0.00)*	0.75 *(-0.06)*	0.78 *(-0.03)*	0.75 *(-0.06)*	0.78 *(-0.03)*	0.81
Average	**0.65 *(-0.19)***	**0.70 *(-0.13)***	**0.81 *(-0.02)***	**0.79 *(-0.04)***	**0.79 *(-0.04)***	**0.70 *(-0.13)***	**0.83**
Program Classification							
`python`	0.53 *(-0.18)*	0.61 *(-0.10)*	0.69 *(-0.02)*	0.65 *(-0.06)*	0.70 *(-0.01)*	0.67 *(-0.04)*	0.71
`powershell.exe`	0.63 *(-0.34)*	0.84 *(-0.13)*	0.74 *(-0.23)*	0.82 *(-0.15)*	0.69 *(-0.28)*	0.91 *(-0.06)*	0.97
`firefox.exe`	0.37 *(-0.33)*	0.41 *(-0.29)*	0.45 *(-0.25)*	0.51 *(-0.19)*	0.41 *(-0.29)*	0.64 *(-0.06)*	0.70
Average	**0.53 *(-0.18)***	**0.61 *(-0.10)***	**0.69 *(-0.02)***	**0.65 *(-0.06)***	**0.70 *(-0.01)***	**0.67 *(-0.04)***	**0.71**

a small portion of an overall provenance graph, so the graph size is an unreliable way to determine if an attack is present in the graph.

Security Domain Features.

Triangles are tight parent-child process interactions, which are crucial in Fileless malware scenarios where a malware is creating the initial infection by dropping and cloning its payload to create multiple copies of itself. However, their simplicity also results in their prevalence in benign programs. In the DARPA datasets, long-running processes created similar amounts of triangles as the attacks. Therefore, it was difficult to reliably differentiate attacks from benign behavior, resulting in agreement with the GAT model being as low as 0.51. Some attacks, such as the `svchost.exe` Fileless malware samples, were frequently identifiable with triangles alone, since the benign program did not create triangles. Empirically, we have seen triangles are commonly a supporting feature that works alongside the more complex structures (*e.g., Exploding Square* and *Jellyfish*).

Squares and Kites form a reliable backbone for explaining anomaly detection and program classification decisions made by the GAT model. Boasting both the highest overall average agreement and top individual performances in the program classification and Fileless Malware detection tasks, squares and kites identify clusters of programs with shared dependencies. The *Kite* pattern is particularly effective at capturing malware replication and deployment because it contains a *Dropper Triangle* as a subgraph, but with the added requirement that the parent and child process share a dependency.

Exploding shapes are specializations of the *Square* and *Kite* patterns that include multiple file read operations by the child malware, and are therefore particularly effective at explaining the GAT 's predictions in the APT dataset from (Mukherjee et al., 2023). These exploding variants are created after the initial deployment, when the final payload is successfully executed. Consequently, the exploding shapes underperformed in the benign program classification task as these are usually seen in equal quantity in the benign context. It is noteworthy that exploding shapes perform very similarly to squares and kites.

Cascade and Jellyfish are used for identifying complex, multi-stage attacks. The archetypical *Cascade* pattern is a sequence of malware payloads with a central process monitoring its progress. The *Jellyfish* pattern simply identifies parent-child processes with many shared dependencies, which is common in parallel processing. These patterns are specialized towards process inheriting and staging behavioral scenarios, resulting in their excellent explanative power on the DARPA dataset, but showing high variability in their effectiveness for Fileless malware samples. This is because these shapes are prevalent in benign service applications (*e.g.,* `sc.exe` and `explorer.exe`) since they are specialized for parallel processing and comprised of multiple child processes that attend to a service request. Therefore, this shape is prevalent in both benign and anomaly samples, resulting in a significant drop in explanative power compared to using the full feature list.

Internal and external network connections shows varying agreement depending on the attack scenario. For instance, in APT, network connections alone are not reliable indicators, as attackers often disguise their activities as legitimate programs that also establish network connections. However, attacks using programs that rarely make network connections, *e.g.,* `reg.exe` or `sc.exe`, become easily identifiable.

All security domain features utilizing the full range of features generally results in the best overall performance. However, there are cases where a subset of features can perform almost as effectively as the complete set. This is evident in the case of `svchost.exe`, where the use of triangles alone achieves an agreement of 0.84, equal to that obtained with all features. This indicates the possibility of optimized feature selection in certain scenarios. Nevertheless, such optimization requires knowledge of the specific attack.

Ablation Study: GraphSAGE The ablation study utilizing the GraphSAGE network revealed patterns consistent with those observed in the GAT network study (§5.8.6), particularly regarding the impact of certain shape groups on PROVEXPLAINER. Notably, the *Square* and *Kite* shapes were distinctively influential for datasets like Fileless Malware. These shapes effectively encapsulate malware replication and deployment processes. Additionally, the *Square* and *Kite* shapes demonstrated notable performance in the APT dataset from (Mukherjee et al., 2023). This effectiveness is attributed to their ability to capture shared dependencies, a vital element in the *initial access* and *establishing a foothold* stages of an attacker's APT campaign—a finding consistent with their performance in the GAT network analysis.

In the context of the DARPA APT dataset, the *exploding shapes* assumed significant importance. This dataset, characterized by numerous long-range dependencies associated with malware, finds an effective representation through the *exploding shapes.* These shapes are particularly adept at capturing scenarios that represent the later stages of an APT campaign, such as *deepen access* and *lateral movement.* These stages are typically marked

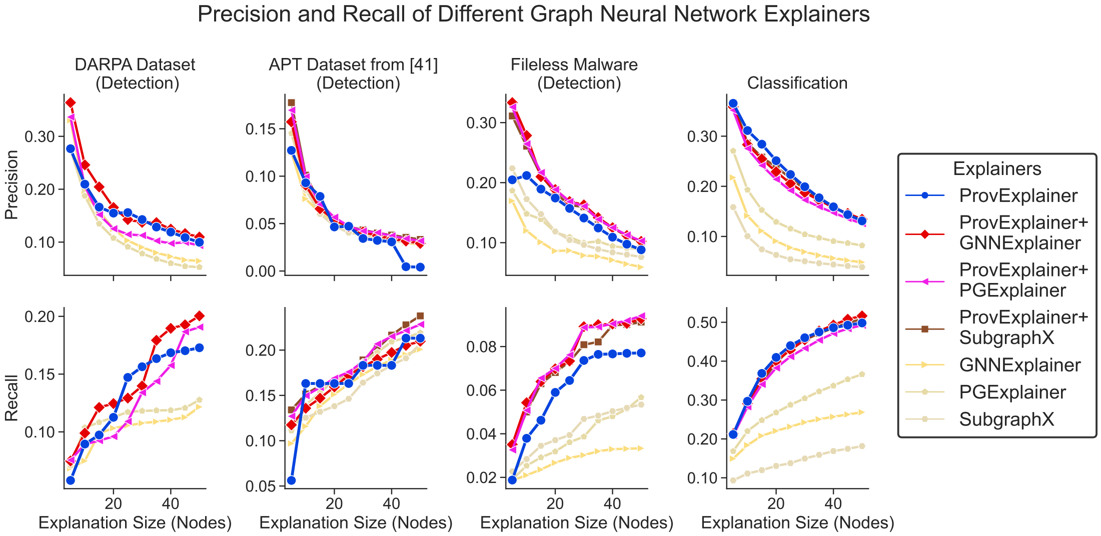


Figure 5.5: Effectiveness of graph model explainers at identifying documented entities (§5.8.1), measured using precision and recall.

by malware activities involving reading several system dependencies or probing system files so that the malware can replicate and spread.

5.8.7 ProvExplainer vs. SOTA Explainers

To answer **RQ2** and **RQ3**, Figure 5.5 compares the precision and recall of explanations derived from SOTA methods (GNNExplainer, PGExplainer, and SubgraphX) with those derived from ProvExplainer. Generally, as we request more nodes from the explanation techniques, the precision trends downwards and the recall trends upwards. ProvExplainer surpasses all existing SOTA explainers across datasets, barring the APT dataset mentioned in (Mukherjee et al., 2023) for explanations exceeding 40 nodes. This exception arises due to ProvExplainer's capture of systemic noise for large window sizes, which impacts precision. Nonetheless, for the APT dataset in (Mukherjee et al., 2023), ProvExplainer excels in generating concise explanations below 40 nodes, aligning with the preferences of security researchers for brief yet comprehensive analyses. Later, we will analyze specific case studies in §5.9.

In the DARPA and APT datasets, the security-aware features of ProvExplainer provide a clear advantage in extracting security-relevant nodes from the provenance graphs. In the APT dataset, we notice a limitation of ProvExplainer, where it can only offer nodes that participate in the defined shapes for the explanation, which can lead to plateaus when there are security-relevant nodes that do not participate in any of the shapes. In the Fileless malware dataset, where the usage of memory objects disrupts several structural patterns, ProvExplainer still provides the best explanations with respect to the documented entities.

Our experimentation showcased an interesting trend among the SOTA explainers: no one SOTA explainer consistently outperformed the rest. GNNExplainer tries to identify substructures that provide maximum mutual information, while PGExplainer generates a probabilistic global model to explain the predictions. The differences in explanation performance across datasets gives insight regarding the data composition depending on if it is easier to create global explanations or local explanations. For the DARPA datasets it is hard to create generalized explanations since the DARPA dataset consists of different APT scenarios that are executed using different payloads and attack tactics. But for the APT dataset from (Mukherjee et al., 2023), which consists of only two different APT scenarios, it is easier to create global explanations, favoring PGExplainer.

More interesting trends emerge when we *combine* explanations (§5.7). Selecting nodes for the explanation according to both ProvExplainer and a general-purpose explainer achieves best or near-best performance with respect to the documented entities across all of our datasets. Particularly in the program classification and Fileless malware detection datasets, even when there is a large gap between two general-purpose explainers, combining them with ProvExplainer improves and stabilizes the performance. As we will further explore in our case studies (§5.9), the different explanation techniques prioritize different aspects of program behavior, causing the composite explanations to be more complete and stable than individual explanations.

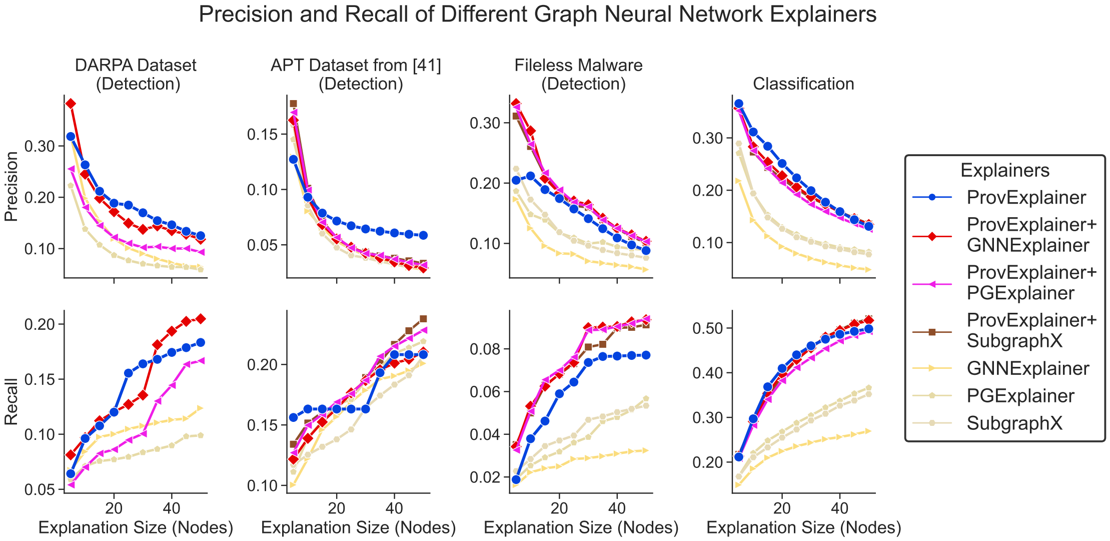


Figure 5.6: Effectiveness of graph model explainers at identifying documented entities (§5.8.1), measured using precision and recall as more nodes are included in the explanation. ProvExplainer outperforms SOTA explainers on anomaly detection tasks and remains competitive in classification tasks.

ProvExplainer vs. SOTA Explainers: GraphSAGE

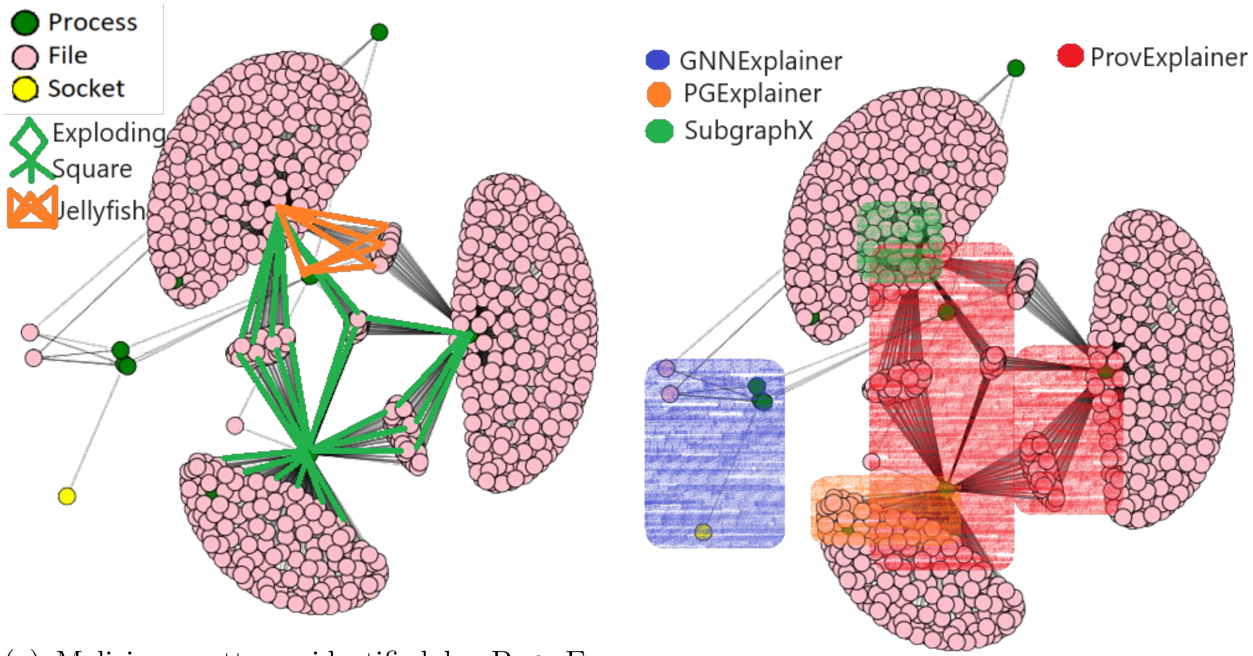
As mentioned previously §5.8.1, we use precision and recall to compare the explanation quality of ProvExplainer and SOTA GNN explainer (GNNExplainer, PGExplainer, and SubgraphX). Similar to GAT based results, ProvExplainer deliver the best performance across the datasets. The complexity (*i.e.,* size and resource interaction) of the provenance graphs play a major role in determining which explainers would be effective for a particular dataset. For smaller provenance graphs like APT graphs from (Mukherjee et al., 2023) PGExplainer performs the second best but for DARPA APT graphs which are known to be complex or noisy GNNExplainer performs the second best. Fileless Malware dataset's malware graph composure is different as compared to APT attacks due to the attack campaigns. So, SubgraphX performed better than both GNNExplainer and PGExplainer. SubgraphX showed its specialization in identifying the distinctive malicious subgraphs which when removed changes the model prediction. However, this specialization also lead to limitations in other scenarios, particularly where the malware is making succinct changes in the system to avoid detection, *e.g.,* APT scenarios.

An interesting trend is that while ProvExplainer performed the best in both the DARPA and APT datasets, the second-best explainer for DARPA is GNNExplainer, while for the APT dataset it is PGExplainer. GNNExplainer tries to identify potentially disjoint substructures that maximize mutual information, but PGExplainer generates a probabilistic global model to explain the predictions. This interesting trend showcases that while GNNExplainer is more suitable when the focus is on understanding specific decisions made by the GNN, and PGExplainer is better suited for scenarios where a global explanation across the dataset is necessary. Since, the two explainers perform at different capacity for the datasets, it gives insight regarding the composition of the dataset *i.e.,* is it easier to create global explanations that are consistent throughout the dataset or is it more apt to create local explanations. For the DARPA datasets it is hard to create generalized explanations since the DARPA dataset consists of different APT scenario that are executed using different payload and attack tactics. But, for the APT dataset which majorly consists of only two different APT scenarios (*e.g.,* Enterprise and Supply-Chain APT), it is easier to create global explanations, so PGExplainer was able to create global explanations that are consistent across datasets.

5.9 Case Studies

To demonstrate ProvExplainer in a realistic setting, we analyze three case studies from the DARPA (DARPA, 2019) datasets. In each study, we use an explanation size of 40 nodes and refer to the GAT model. We list the most important features from the surrogate DT and qualitatively analyze the explanations from ProvExplainer and the SOTA GNN explainers. Detailed system-level analyses can be found in the appendix (§5.9).

5.9.1 FiveDirections: Browser Extension



(a) Malicious patterns identified by ProvExplainer.

(b) GNN explainations comparison.

Figure 5.7: FiveDirections: the attacker exploits the target via a malicious `Firefox` extension.

Description. An attacker targets the `Firefox` browser using a malicious extension to deploy the `drakon` malware. The attacker writes `drakon` directly to disk and exploits a compromised browser extension masquerading as a password manager to execute malicious `powershell` code, gaining deeper access and control over the system, as illustrated in Figure 5.7a.

Features: *Exploding Square* and *Jellyfish*.

System Interpretation. This attack graph contains the *Jellyfish* and *Exploding Square* patterns, which are often observed in the APT stages of *deepen access* (ATT&CK®, 2020) and *lateral movement* (ATT&CK®, 2018c). The *Jellyfish* pattern captures the dependency correlation among malware processes, where multiple instances of `drakon` exploit `Firefox` vulnerabilities via a malicious extension to spread to different parts of the system. This pattern is emblematic of malware processes cloning themselves to persist in the system as well as maintain operational integrity. Further, the *Exploding Square* highlights how malware processes move laterally to successfully exfiltrate sensitive data and read configuration files.

System Connection. In the context of the described attack, where the `drakon` malware exploits the `firefox.exe` browser through a rogue extension (`pass_mgr`) the *Jellyfish* shape is created. Multiple instances of the malware process are created, each reading from the malicious files: `pass_mgr.exe` and `passwordfile.dat`. Additionally, they access essential dictionary files (`en-US.aff`) and cryptographic libraries (`bcryptprimitives.dll`) to maintain operational consistency, allowing them to conduct their malicious activities efficiently. The *Exploding Square* shape captures the data extraction behavior through sensitive file accesses. These malware access sensitive information and system configuration files like `WindowsShell.Manifest`, `shell32.dll`, `cryptbase.dll`, and `wintrust.dll`, along with initial malware files (`addons`, `tzres.dll`, `userenv.dll`).

GNNExplainer effectively identifies the malware template file present in `C:\ProgramFiles\MozillaFirefox\add-on\pass_mgr.exe` and the initial malware `pass_mgr.exe`. PGExplainer recognized the structures common across all attack graphs, particularly identifying file access of system libraries needed for malware operation `C:\*\System32\driver`, `C:\*\Windows\SysWOW64` and `C:\*\AppData\Local\Temp`. SubgraphX performed at the same capacity as PGExplainer as it identified a different the structure of malware executing its payload from `C:\*\Desktop\*\add-on`, reading sensitive files from `C:\*\ProgramFiles\MozillaFirefox\*`, and extracting them through `C:\*\admin\AppData\*`. This is partly due to its foundation on Monte Carlo Tree Search (MCTS), incorporating nondeterministic exploitation and exploration stages. In the absence of attribute information in the GNN, the exploitation stage lacks guidance. But, when SubgraphX correctly identifies a substructure for exploitation, the exploration stage of MCTS proves effective.

ProvExplainer vs. SOTA explainers. Figure 5.7b compares the explanations of ProvExplainer and those of SOTA GNN explainers, focusing on their efficacy in identifying security-aware elements. ProvExplainer excels by highlighting the malware replicating itself from template files and accessing sensitive system files. ProvExplainer isolates security-relevant structures in the graph, significantly increasing the end-user trust in the detection.

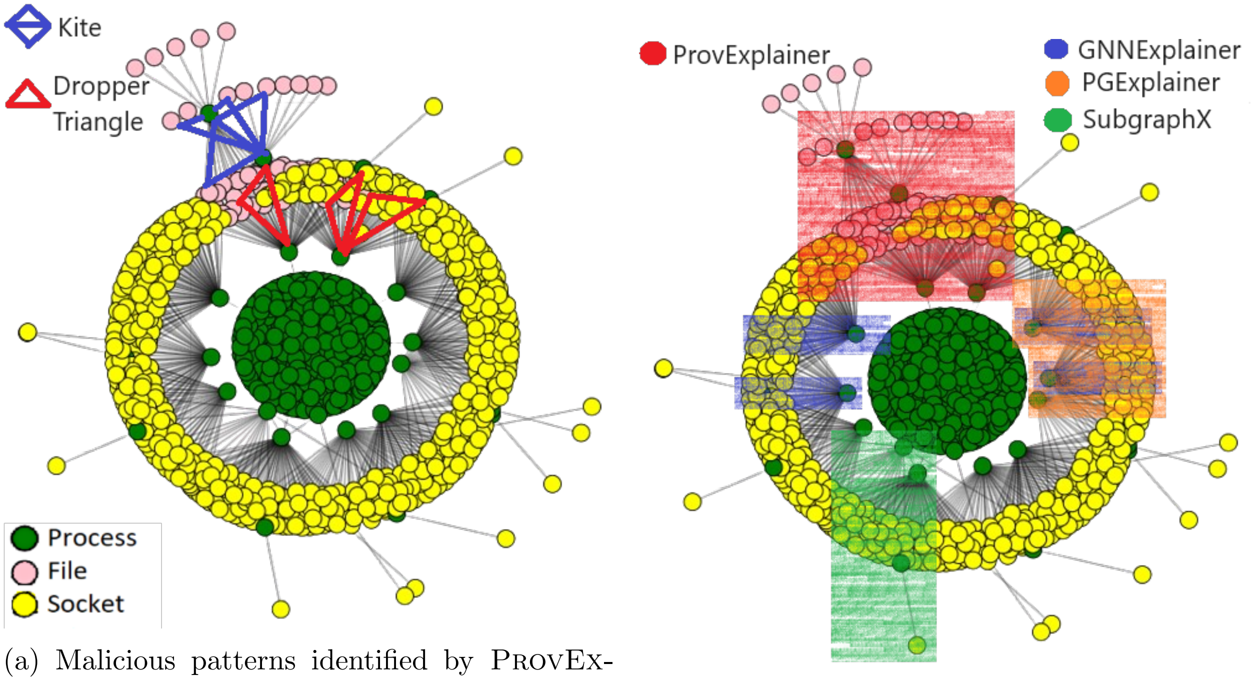
GNNExplainer identifies the malware template file and the malicious extension. This effectiveness stems from GNNExplainer's method of searching for important edges. When this GNNExplainer isolates the pivot structure, the graph becomes disjoint, leading to a change in prediction. This results in a high information gain, the core metric for GNNExplainer. On the other hand, PGExplainer and SubgraphX reveal commonalities across attack graphs, such as the identification of key system library accesses required for malware operation. SubgraphX's effectiveness varies due to its Monte Carlo Tree Search (MCTS), suggesting benefits to incorporating domain-specific insights into SubgraphX's scoring function.

5.9.2 FiveDirections: Copykatz

Description. In a sophisticated attack, a hijacked version of usdoj.gov exploits `Firefox` to deploy `drakon` malware to the victim host. Then, `drakon` uses the *elevate* driver to escalate privileges and masquerade as the `runtimebroker` system program. Finally, the malicious `runtimebroker` instance connects to a command and control (C2) server to download and execute `Copykatz` (an older version of `Mimikatz`) to harvest and exfiltrate host credentials.

Features: *Dropper Triangle* and *Kite*.

System Interpretation. The attack graph shown in Figure 5.8a highlights two key patterns: the *Dropper Triangle* and the *Kite*. The *Dropper Triangle* captures the initial access by highlighting the creation of malicious dynamic-link libraries. This stage enables the Windows Application Programming Interface and cryptographic operations necessary for the malware's functionality. Following this, the malware disguises itself as `Firefox`, and executes the `Copykatz` payload.



(a) Malicious patterns identified by ProvExplainer.

(b) GNN explainations comparison.

Figure 5.8: FiveDirections: the attacker gains C2 connections and installs `Copykatz` through a `Firefox` exploit.

The malicious `Firefox` instance then distributes its payload through temporary files, setting the stage for a subsequent malicious `Firefox` instance to trigger a flood of process creations. This chain of actions, marked by inherited functionalities from `Copykatz`, forms the *Kite* pattern. The replication of this malware leverages the same essential system libraries, indicating a meticulous design to maintain operational consistency throughout the malicious process chain.

System Connection. The *Dropper Triangle* starts its kill chain stage of initial accessc (ATT&CK®, 2018b) by reading shared dynamic-link libraries (DLLs); notable mentions are `ntdll.dll` and `bcryptprimitives.dll`. It then writes and executes a malware masquerading as `firefox.exe` which contains the `Mimikatz` and `Copykatz` modules. `ntdll.dll` (ntd, 2018) includes multiple kernel-mode functions which enables the "Windows Application Programming Interface (API)" and `bcryptprimitives.dll` (bcr, 2018) contain functions implementing cryptographic primitives, which are essential for `Mimikatz` (mim, 2018) and `Copykatz` to function.

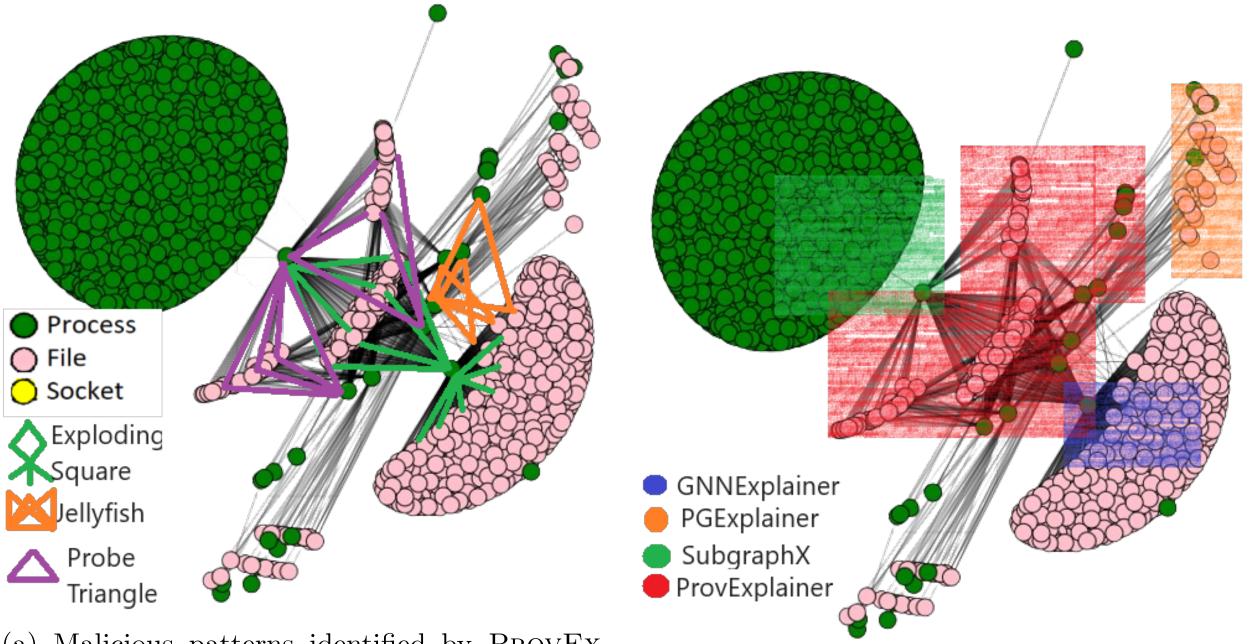
The `firefox.exe` malware write its payload into temporary files, such as `virtuous` and `tropical`. Subsequently, another malicious instance of the `firefox.exe` malware initiates a domino effect by creating a chain of processes by executing the malware template. This dependency correlation between the malware and its parent is characterized by functional inheritance, where the children require the same system library files as the parent to function correctly. The DLLs involved are read from `ProgramFiles` and `System32`.

GNNExplainer and PGExplainer correctly identified the data extraction stage where four malicious `firefox.exe` processes make C2 connections to the external IPs (202.179.137.58 and 217.160.205.44). `firefox.exe` was invoked by malware masquerading as `runtimebroker.exe`, so if GNNExplainer masks out the process creation edge of the children `firefox.exe`, that would lead to a graph cut with two separate graphs, leading to a change in the prediction. SubgraphX also highlighted similar `firefox.exe` processes that are created by the first `runtimebroker.exe`, but instead of making external C2 connection it created another `firefox.exe` process for rendering content from localhost (127.0.0.1).

ProvExplainer vs. SOTA Explainers. GNNExplainer and PGExplainer pinpoint the stage where multiple `Firefox` processes connect to the C2 servers. This activity traces back to the malicious `runtimebroker` instance. Notably, SubgraphX detects an alternate trajectory where a `Firefox` process, instead of reaching out to external servers, spawns another process aimed at local content manipulation, showcasing the malware's versatility in engaging with both external and internal resources for its objectives.

While SOTA explainers have demonstrated proficiency in identifying the final stages of this data breach, only ProvExplainer effectively captured both the initial infection and its propagation. This distinction underscores the importance of recognizing early-stage indicators for root cause analysis.

5.9.3 Trace: Phishing E-mail



(a) Malicious patterns identified by ProvExplainer.

(b) GNN explainations comparison.

Figure 5.9: Trace: after an employee clicks on a phishing link, `Firefox` installs multiple Trojans to exfiltrate sensitive data.

Description. The attacker first launches a phishing campaign to compromise the identity of an employee. Leveraging the employee's identity, the attacker then targets other employees with deceptive emails containing links to a malicious website. This website installs a Trojan in the victims' computers, which then creates multiple copies of itself, overflowing the system. These cloned Trojans read sensitive user files while the original Trojan achieves persistence in the system.

Features: *Probe Triangle*, *Exploding Square*, and *Jellyfish*.

System Interpretation. In the Trace APT scenario, the *Probe Triangle*, *Exploding Square*, and *Jellyfish* patterns elucidate the malware's structure (Figure 5.9a). The *Probe Triangle* reveals the Trojan's cloning activity, where it is downloaded and replicates itself to establish a foothold. By masquerading as benign programs, the malware disguises its malicious processes, allowing it to proliferate undetected.

The Trojan, after establishing itself, impersonates system processes to execute a malicious script, leading to the creation of multiple copies. The *Jellyfish* pattern illustrates the dependency correlation among the cloned malware instances, interacting with similar system configuration files to operate efficiently as well as constructing a detailed profile of the target system. Ultimately, the extraction of data from sensitive system files is captured by the *Exploding Square*. This extraction is part of a larger scheme of lateral movement by profiling the available system processes, enabling the malware to leverage system libraries effectively.

System Connection. The *Probe Triangle* captured the staging behavior of the Trojan. The Trojan was downloaded from www.nasa.ng, executed and replicated itself within the system. Specifically, the malware named `nasa.ng` is placed in `/home/admin/.mozilla/firefox/` and `/usr/local/firefox-54.0.1/obj-x86_64-pc-linux-gnu/`. The Trojan created new malware with benign names such as `firefox` to effectively evade detectors, to replicate unhindered and overloaded the system with malicious processes. After the malware successfully staged, the malware masquerading as `/bin/sh` to read the malicious script staged in (`/etc/update-motd.d/00-header/`) and executed it to create multiple copies of itself. The malware reads various system configuration files present in `/etc/protocols/`, `/etc/lsb-release/`, and `/etc/hosts.deny/`. Reading sensitive system configuration files are essential to build the system profile. The *Jellyfish* shape captured the dependency correlation of the malwares being created that inherited similar configuration.

Ultimately, the malware completes its target of reading sensitive system files from `/etc/fonts/conf.d/`, `/usr/lib/x86_64-linux-gnu/`, and `/usr/share/X11/local/`. These activities are aimed at gathering system information to create a profile of the company and the devices in use. The attacker wants to create a profile of the victim environment to ensure their malware can effectively leverage system libraries to complete their objective. There is an overlap in the files (present in `\etc\hosts` and `/usr/lib/x86_64-linux-gnu/*`) involved in the *Probe Triangle* and *Jellyfish* operations because the malware replicates itself probing and inheriting the functional dependencies of its parent.

GNNExplainer was able to correctly capture the staging behavior where the malware from `nasa.ng` read shared library (`/usr/lib/x86_64-linux-gnu/*.so.*`) and cache file (`/usr/share/applications/mimeinfo.cache`, `/usr/lib/x86_64-linux-gnu/*/loaders.cache`). PGExplainer incorrectly indicated benign substructures, but SubgraphX correctly captured the inheritance behavior of `firefox.exe` executing multiple times with the argument file http://www.nasa.ng/, to create the malware clones from the template. SOTA explainers missed the malware's ultimate goal of reading sensitive files, which was only captured by ProvExplainer.

ProvExplainer vs. SOTA explainers. GNNExplainer identifies the initial malware staging behaviors, including interactions with shared libraries and cache files. Meanwhile, PGExplainer captures the benign structure of `xfce4-appfinder`—a lightweight desktop environment for UNIX systems, invoked by a DARPA script to simulate an enterprise environment. SubgraphX identifies the malicious inheritance behavior of `Firefox` executing a template to generate multiple malware clones. However, it overlooks sensitive file accesses, a detail exclusively captured by ProvExplainer. ProvExplainer comprehensively traced the malware kill chain from payload deployment and clone creation to the final step of accessing sensitive files.

CHAPTER 6

FUTURE WORK AND CONCLUSION

System provenance is a vast research area with many unsolved problems. With the contribution of this dissertation, certainly, we have not solved all the challenges. However, this is a step towards solving three long-standing provenance domain problems: scalability in computationally limited IoT environments, resilience against adversarial manipulation, and interpretability of security alerts. In the following section, we discuss the current limitations of this dissertation and our thoughts on solving them.

6.1 Future Research Directions

6.1.1 Real-Time Prevention

Although we focus on a detection in this dissertation, ProvIoT can be easily extended to provide real-time prevention (*e.g.,* blocking or killing anomalous processes). ProvIoT can also be augmented with other triaging and defense mechanisms (*e.g.,* dynamic quarantine or deep inspection) when it raises alerts. ProvIoT supports online forensic analysis including backtracking analysis and data query by leveraging its extensive system event collection.

6.1.2 Automated Evasive Attack Generation

Currently, the attack vectors generated by ProvNinja must be implemented manually, which is a labor-intensive process that requires significant domain knowledge and technical expertise. If ProvNinja were integrated with attack implementation frameworks (Metasploit, 2021; ATT&CK®, 2022b), it would dramatically reduce the required domain expertise, resource overhead, and time cost to construct real attack chains to either deploy offensively or to use as defensive training exercises. Further, such an integration would necessarily be aware of the security implications of each process transition, and therefore be capable of automatically preserving attack semantics.

6.1.3 Defense Against Evasive Attacks

Despite approaching the task from the adversary's perspective, ProvNinja research will help defenders by providing a tool to generate potential attack sequences to harden their security models. ML detectors, when trained in an adversarial way against these evasive attacks, can uncover events that are either robust against evasive modification or crucial components of the attacks. Additionally, ProvNinja only focuses on improving the one-hop likelihood of events in the malicious chains; by overcoming the challenge of focusing on long-range casual dependencies, defenders can unmask the anomalies induced by the malicious behavior. We leave dedicated defense model training against evasive mimicry attacks as important future work.

For defenders, natively integrating ProvNinja with a adversarial training framework for provenance-based ML detectors will help defense models pinpoint the subtle differences between regular users and stealthy attackers. Because defenders can perform the strongest form of ProvNinja with access to the true target dataset, defense models that perform well in this training are also likely to detect ProvNinja-style APT attacks in the wild.

6.1.4 Environmental Dependencies of Evasive Attacks

Unlike traditional ML research where the problem space is similar to the feature space (*e.g.,* image processing), it is impossible to fully replicate the victim's problem space environment (target system with normal concurrent user activity). An adversarial attack generated with the attacker's environment will therefore typically differ from an attack generated with the real victim environment. The quantification of ProvNinja's sensitivity to environmental differences would help gauge the practicality of its application in the wild.

6.1.5 Adversarial Manipulation of Graph Features

We look at what are the implications of attackers using knowledge of the graph structural features to design adversarial attacks against the GNN-based IDS explainer. The explainer models used in ProvExplainer is separate from the black-box detection model. Fooling both the interpretable surrogates and the underlying black-box model is the ideal case for an attacker seeking to avoid suspicion, and an attacker who can attack the explainer this way can also attack the detection model directly (Goyal et al., 2023; Mukherjee et al., 2023). Creating robust detection and explanation systems that can withstand adversarial manipulation is a critical open research problem, and is beyond the scope of ProvExplainer.

6.1.6 Support for Fine-Grained Detection Tasks

Recent developments in provenance-based security detection systems have trended towards fine-grained node and edge level anomaly detection (Rehman et al., 2024; Zengy et al., 2022). The explanation requirements of these detectors differ from those of whole-graph anomaly detectors and classifiers. In contrast to using explanations to narrow down the context for consideration by security practitioners, fine-grained models will need explainers to bring in relevant context to aid human analysis. Providing security-aware explanation to advanced fine-grained security detectors is an exciting and important direction for future work.

6.2 Conclusion

In this dissertation, we have illustrated the importance of scalable, robust, and explainable provenance-based intrusion detection systems (PIDS) in the area of infrastructure security. We demonstrated how we solved three key challenges in system provenance analysis.

First, we presented ProvIoT, a novel end-to-end edge-cloud collaborative security framework for IoT devices. ProvIoT adapts modern provenance-based anomaly detection for IoT environments, performing on-device training and detection using federated learning. This approach ensures privacy for local system events while maintaining high detection accuracy and low resource overhead. Through extensive evaluation with a realistic provenance dataset, ProvIoT demonstrated exceptional performance, detecting Fileless malware and APT attacks. Additionally, ProvIoT incurs minimal resource overhead, and operates effectively without requiring continuous network connectivity. These results highlight the framework's suitability for robust and privacy-preserving detection in resource-constrained IoT devices.

Next, we introduced ProvNinja, a data-driven evasive attack generation framework designed to exploit vulnerabilities in PIDS by replicating the behavioral patterns of benign system programs. Our research is the first to explore the design space of evasive attack generation, overcoming the challenge of bridging the gap between system actions and their feature space representations. ProvNinja successfully generated and deployed evasive attacks against multiple ML detectors, demonstrating significant reduction in defense model F1 scores. These attacks are not only theoretical but are actualized in real-world environments, posing a significant threat to existing PIDS models. This work provides valuable insights into the weaknesses of PIDS and establishes ProvNinja as a practical tool for evaluating the resilience of security models against evasive threats.

Finally, we introduced ProvExplainer, a framework that enhances transparency and accountability in PIDS by defining security-aware graph structural features with corresponding system-level interpretations. By leveraging these interpretable features, ProvExplainer successfully approximates decisions made by complex GNN on tasks such as APT detection, Fileless malware detection, and program classification. Our case studies, using real-world APTs, showed that ProvExplainer improves the explainability of GNN-based PIDS systems, over SOTA GNN explainers. Furthermore, the combination of SOTA GNN explainers with ProvExplainer provided more stable and complete explanations, leading to an additional increase in explainability metrics. This framework marks a significant advancement toward achieving transparency and trustworthiness in PIDS decision-making.

In conclusion, this dissertation has demonstrated the critical role of system provenance in addressing modern security challenges, particularly in resource-constrained environments like IoT devices, the generation of adversarial attacks to test the robustness of PIDS models, and the development of interpretable frameworks to ensure transparency in GNN-based security systems. Together, these contributions represent a significant step towards the evolution of scalable, robust, and explainable provenance-based intrusion detection systems (PIDSs).

REFERENCES

(2015). Apple watch ram size comparison chart: How much ram does apple watch have? https://www.knowyourmobile.com/wearable-technology/apple-watch-ram-size/. (Accessed on 05/26/2023).

(2015). Google nest - support. https://support.google.com/googlenest/answer/9230098. (Accessed on 05/26/2023).

(2017). Google home mini teardown, comparison to echo dot, and giving technology a voice. https://tinyurl.com/ykbay2fu. (Accessed on 05/26/2023).

(2018). Bcryptprimitives.dll. http://tinyurl.com/yvpesvzt. (Accessed on 01/21/2024).

(2018). Mimikatz. http://tinyurl.com/3styvesw. (Accessed on 01/21/2024).

(2018). Ntdll.dll. http://tinyurl.com/yc2z88px. (Accessed on 01/21/2024).

(2018). Raspberry Pi – Teach, Learn, and Make with Raspberry Pi. https://www.raspberrypi.org.

(2019). Cuckoo sandbox. https://tinyurl.com/33jdwr93. Accessed: April 6, 2023.

(2019). Trojan.win32.scar.ad. https://tinyurl.com/3sdj642z.

(2020). Inside amazon's ring alarm system. https://tinyurl.com/yck5jm4m. (Accessed on 05/26/2023).

(2020). Suricata. https://suricata.io/. Accessed: April 6, 2023.

(2020). Yara. https://virustotal.github.io/yara/. Accessed: April 6, 2023.

(2021). Cyber kill chain® — lockheed martin. https://www.lockheedmartin.com/en-us/capabilities/cyber/cyber-kill-chain.html. (Accessed on 07/24/2021).

(2021). Nssm - the non-sucking service manager alternatives. https://tinyurl.com/2p8n5kca. Accessed: April 6, 2023.

(2021). Snort. https://www.snort.org/. Accessed: April 6, 2023.

(2022). Cve-2022-21882. https://tinyurl.com/2p9cmftm. Accessed: April 6, 2023.

(2022). Smart refrigerator with family hub. https://tinyurl.com/4kz6z6z5. (Accessed on 05/26/2023).

(2023a). Canvas. http://tinyurl.com/y4wrf74u.

(2023b). Canvas. http://tinyurl.com/yaknev56.

Acar, A., H. Fereidooni, T. Abera, A. K. Sikder, M. Miettinen, H. Aksu, M. Conti, A.-R. Sadeghi, and S. Uluagac (2020). Peek-a-boo: I see your smart home activities, even encrypted! In *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '20, New York, NY, USA, pp. 207–218. Association for Computing Machinery.

Ahmad, A., S. Lee, and M. Peinado (2022, 5). HARDLOG: Practical Tamper-Proof SystemAuditing Using a Novel Audit Device. In *Security and Privacy (SP)*, Security and Privacy (SP).

ATT&CK®, M. (2017). Data from local system. https://attack.mitre.org/techniques/T1005/.

ATT&CK®, M. (2018a). Exfiltration. https://attack.mitre.org/tactics/TA0010/.

ATT&CK®, M. (2018b). Exploitation for client execution. hhttp://tinyurl.com/muhzctfb.

ATT&CK®, M. (2018c). System information discovery. https://attack.mitre.org/techniques/T1082/.

ATT&CK®, M. (2020). Create or modify system process. https://attack.mitre.org/techniques/T1543/.

ATT&CK®, M. (2021a). Compromise client software binary. https://attack.mitre.org/techniques/T1554/. Accessed on 11/29/2021.

ATT&CK®, M. (2021b). Data collection. https://attack.mitre.org/tactics/TA0004/. Accessed: April 6, 2023.

ATT&CK®, M. (2021c). Data manipulation. ATT&CK. Accessed: April 6, 2023.

ATT&CK®, M. (2021d). gsecdump. https://attack.mitre.org/software/S0008/. Accessed: April 6, 2023.

ATT&CK®, M. (2021e). Process injection: Dynamic-link library injection. https://attack.mitre.org/techniques/T1534/. Accessed: April 6, 2023.

ATT&CK®, M. (2021f). Process injection: Ptrace system calls, sub-technique t1055.008 - enterprise — mitre att&ck®. https://attack.mitre.org/techniques/T1055/008/. (Accessed on 07/23/2021).

ATT&CK®, M. (2022a). Initial access: Tactics, techniques, and procedures. https://attack.mitre.org/tactics/TA0001/. Accessed: April 6, 2023.

ATT&CK®, M. (2022b). Mitre att&ck®. https://attack.mitre.org/. Accessed: April 6, 2023.

Avllazagaj, E., Z. Zhu, L. Bilge, D. Balzarotti, and T. Dumitras (2021). When malware changed its mind: An empirical study of variable program behaviors in the real world. In *USENIX Security Symposium (SEC).*

Bahşi, H., S. Nõmm, and F. B. La Torre (2018). Dimensionality reduction for machine learning based iot botnet detection. In *2018 15th International Conference on Control, Automation, Robotics and Vision (ICARCV)*, pp. 1857–1862. IEEE.

Bansal, A., A. Kandikuppa, C.-Y. Chen, M. Hasan, A. Bates, and S. Mohan (2022). Towards efficient auditing for real-time systems. In *Computer Security–ESORICS 2022: 27th European Symposium on Research in Computer Security, Copenhagen, Denmark, September 26–30, 2022, Proceedings, Part III*, pp. 614–634. Springer.

Barbero, F., F. Pendlebury, F. Pierazzi, and L. Cavallaro (2022). Transcending transcend: Revisiting malware classification in the presence of concept drift. In *IEEE Symposium on Security and Privacy (SP).*

Bareckas, K. (2022). The mydoom worm: history, technical details, and defense. https://nordvpn.com/blog/mydoom-virus/.

Barr-Smith, F., X. Ugarte-Pedrero, M. Graziano, R. Spolaor, and I. Martinovic (2021). Survivalism: Systematic Analysis of Windows Malware Living-Off-The-Land. In *IEEE Symposium on Security and Privacy (SP).*

Bostani, H. and M. Sheikhan (2017). Hybrid of anomaly-based and specification-based ids for internet of things using unsupervised opf based on mapreduce approach. *Computer Communications 98*, 52–71.

Breunig, M. M., H.-P. Kriegel, R. T. Ng, and J. Sander (2000). Lof: Identifying density-based local outliers.

Carlini, N. (2019). A complete list of all (arxiv) adversarial example papers. https://tinyurl.com/3cvur5j7. Accessed: April 6, 2023.

Chaudhary, A., H. Mittal, and A. Arora (2019). Anomaly detection using graph neural networks. In *International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon).*

Chawathe, S. S. (2018). Monitoring iot networks for botnet activity. In *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*, pp. 1–8. IEEE.

Cheng, Z., Q. Lv, J. Liang, Y. Wang, D. Sun, T. Pasquier, and X. Han (2024). Kairos: Practical intrusion detection and investigation using whole-system provenance. In *IEEE Symposium on Security and Privacy (SP)*.

Cosson, A., A. K. Sikder, L. Babun, Z. B. Celik, P. McDaniel, and A. S. Uluagac (2021). Sentinel: A robust intrusion detection system for iot networks using kernel-level system information. In *Proceedings of the International Conference on Internet-of-Things Design and Implementation*, pp. 53–66.

Costin, A. and J. Zaddach (2019). IoT Malware: Comprehensive Survey, Analysis Framework and Case Studies. BlackHat Briefings.

Cozzi, E., M. Graziano, Y. Fratantonio, and D. Balzarotti (2018a). Understanding linux malware. In *2018 IEEE symposium on security and privacy (SP)*, pp. 161–175. IEEE.

Cozzi, E., M. Graziano, Y. Fratantonio, and D. Balzarotti (2018b). Understanding linux malware. In *IEEE Symposium on Security and Privacy (SP)*.

Crowdstrike (2018). Wannamine cryptomining: Harmless nuisance or disruptive threat? http://tinyurl.com/ycxvukjk.

CrowdStrkie (2020). ENDPOINT DETECTION AND RESPONSE (EDR). Technical report, CrowdStrkie.

Cybersecurity, A. (2021, Jan). Malware using new ezuri memory loader — at&t alien labs. https://cybersecurity.att.com/blogs/labs-research/malware-using-new-ezuri-memory-loader. (Accessed on 07/23/2021).

DARPA. Ta5.1 ground truth report engagement 3. https://drive.google.com/file/d/1mrs4LWkGk-3zA7t7v8zrhm0yEDHe57QU.

DARPA (2019). Transparent computing engagement 5. https://github.com/darpa-i2o/Transparent-Computing/blob/master/README-E3.md.

Demontis, A., M. Melis, M. Pintor, M. Jagielski, B. Biggio, A. Oprea, C. Nita-Rotaru, and F. Roli (2019, 8). Why do adversarial attacks transfer? explaining transferability of evasion and poisoning attacks. In *USENIX Security Symposium (SEC)*.

DGL (2022). Deep graph library: Easy deep learning on graphs. https://www.dgl.ai/. (Accessed on 09/21/2021).

die.net (2017). Linux man page. https://linux.die.net/man/.

Ding, F. (2017). Iot malware. https://github.com/ifding/iot-malware.

Ding, F., H. Li, F. Luo, H. Hu, L. Cheng, H. Xiao, and R. Ge (2020). Deeppower: Non-intrusive and deep learning-based detection of iot malware using power side channels. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, pp. 33–46.

Eddy, N. (2024). Ukraine military targeted with russian apt powershell attack. https://shorturl.at/Bdx7x.

Fang, P., P. Gao, C. Liu, E. Ayday, K. Jee, T. Wang, Y. F. Ye, Z. Liu, and X. Xiao (2022, 8). Back-Propagating System Dependency Impact for Attack Investigation. In *USENIX Security Symposium (SEC)*.

Fei, P., Z. Li, Z. Wang, X. Yu, D. Li, and K. Jee (2021, 8). SEAL: Storage-efficient Causality Analysis on Enterprise Logs with Query-friendly Compression. In *USENIX Security Symposium (SEC)*.

Fernandes, E., J. Jung, and A. Prakash (2016). Security Analysis of Emerging Smart Home Applications. In *IEEE S&P*.

FireEye (2020). Evasive attacker leverages solarwinds supply chain compromises with sunburst backdoor. https://tinyurl.com/bdz8s5yn.

Forrest, C. (2017). Iot is a gold mine for hackers using fileless malware for cyberattacks. https://tinyurl.com/ytnmhax8. Accessed: April 6, 2023.

Ganz, T., P. Rall, M. Härterich, and K. Rieck (2023). Hunting for truth: Analyzing explanation methods in learning-based vulnerability discovery. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*.

Google (2018). Google Assistant, your own personal Google. https://assistant.google.com/.

Google (2021a). Edge tpu - run inference at the edge — google cloud. https://cloud.google.com/edge-tpu. (Accessed on 07/23/2021).

Google (2021b). Intro to autoencoders.

Goyal, A., X. Han, G. Wang, and A. Bates (2023). Sometimes, you aren't what you do: Mimicry attacks against provenance graph host intrusion detection systems. In *Network and Distributed System Security Symposium (NDSS)*.

Goyal, A., G. Wang, and A. Bates (2024). R-caid: Embedding root cause analysis within provenance-based intrusion detection. In *IEEE Symposium on Security and Privacy (SP)*.

Grammatikakis, K. P., I. Koufos, N. Kolokotronis, C. Vassilakis, and S. Shiaeles (2021). Understanding and mitigating banking trojans: From zeus to emotet. In *IEEE International Conference on Cyber Security and Resilience (CSR)*.

Guo, W., D. Mu, J. Xu, P. Su, G. Wang, and X. Xing (2018, 11). Lemna: Explaining deep learning based security applications. In *ACM conference on Computer and Communications Security (CCS)*.

Hamilton, W., Z. Ying, and J. Leskovec (2017). Inductive representation learning on large graphs.

Han, X., T. Pasquier, A. Bates, J. Mickens, and M. Seltzer (2020). UNICORN: Runtime Provenance-Based Detector for Advanced Persistent Threats. In *Network and Distributed System Security Symposium (NDSS)*.

Han, X., X. Yu, T. Pasquier, D. Li, J. Rhee, J. Mickens, M. Seltzer, and H. Chen (2021). Sigl: Securing software installations through deep graph learning. In *USENIX Security Symposium (SEC)*.

Harpaz, O. (2020). Fritzfrog: A new generation of peer-to-peer botnets - guardicore. https://bit.ly/3mJzyeq. (Accessed on 07/23/2021).

Hassan, W. U., S. Guo, D. Li, Z. Chen, K. Jee, Z. Li, and A. Bates (2019). NoDoze: Combatting Threat Alert Fatigue with Automated Provenance Triage. In *Network and Distributed System Security Symposium (NDSS)*.

Herath, J. D., P. P. Wakodikar, P. Yang, and G. Yan (2022). Cfgexplainer: Explaining graph neural network-based malware classification from control flow graphs. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*.

Hossain, M. N., S. M. Milajerdi, J. Wang, B. Eshete, R. Gjomemo, R. Sekar, S. Stoller, and V. Venkatakrishnan (2018). Sleuth: Real-time attack scenario reconstruction from cots audit data. In *USENIX Security Symposium (SEC)*.

Hossain, M. N., S. Sheikhi, and R. Sekar (2020, 05). Combating Dependence Explosion in Forensic Analysis Using Alternative Tag Propagation Semantics. In *IEEE Symposium on Security and Privacy (SP)*.

Inam, M. A., Y. Chen, A. Goyal, J. Liu, J. Mink, N. Michael, S. Gaur, A. Bates, and W. U. Hassan (2023). SoK: History is a Vast Early Warning System: Auditing the Provenance of System Intrusions. In *IEEE Symposium on Security and Privacy (SP)*.

Jacobs, A. S., R. Beltiukov, W. Willinger, R. A. Ferreira, A. Gupta, and L. Z. Granville (2022, 11). Ai/ml for network security: The emperor has no clothes. In *ACM Conference on Computer and Communications Security (CCS)*.

Jha, M., C. Seshadhri, and A. Pinar (2015). Path sampling: A fast and provable method for estimating 4-vertex subgraph counts. In *Proceedings of the 24th international conference on world wide web*.

Jia, Y. J., Q. A. Chen, S. Wang, A. Rahmati, E. Fernandes, Z. M. Mao, and A. Prakash (2017). ContexIoT: Towards Providing Contextual Integrity to Appified IoT Platforms. In *NDSS*.

Jia, Z., Y. Xiong, Y. Nan, Y. Zhang, J. Zhao, and M. Wen (2024). Magic: Detecting advanced persistent threats via masked graph representation learning. In *USENIX Security Symposium (SEC)*.

Kaspersky (2020). Fileless threats protection. https://tinyurl.com/vbb9xk47. Accessed: April 6, 2023.

King, S. T. and P. M. Chen (2003a). Backtracking intrusions. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*.

King, S. T. and P. M. Chen (2003b). Backtracking intrusions. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

Kipf, T. N. and M. Welling (2016). Variational graph auto-encoders.

kodi (2018). Kodi — Open Source Home Theater Software. https://kodi.tv/.

Kolda, T. G., A. Pinar, T. Plantenga, C. Seshadhri, and C. Task (2014). Counting triangles in massive graphs with mapreduce. *SIAM Journal on Scientific Computing*.

Kosan, M., S. Verma, B. Armgaan, K. Pahwa, A. Singh, S. Medya, and S. Ranu (2023). Gnnx-bench: Unravelling the utility of perturbation-based gnn explainers through in-depth benchmarking.

Küchler, A., A. Mantovani, Y. Han, L. Bilge, and D. Balzarotti (2021). Does every second count? time-based evolution of malware behavior in sandboxes. In *Network and Distributed System Security Symposium (NDSS)*.

Kumar, A. and T. J. Lim (2019). Early detection of mirai-like iot bots in large-scale networks through sub-sampled packet traffic analysis. *arXiv preprint arXiv:1901.04805*.

Kurakin, A., I. Goodfellow, and S. Bengio (2016). Adversarial machine learning at scale. *arXiv preprint arXiv:1611.01236*.

Lamport, L., R. Shostak, and M. Pease (1982, July). The byzantine generals problem. *ACM Trans. Program. Lang. Syst. 4*(3), 382–401.

Le, Q. and T. Mikolov (2014). Distributed representations of sentences and documents. In *International conference on machine learning*, pp. 1188–1196.

Li, Z., Q. A. Chen, R. Yang, Y. Chen, and W. Ruan (2021). Threat detection and investigation with system-level provenance graphs: a survey. *Computers & Security 106*, 102282.

Liu, Y., M. Zhang, D. Li, K. Jee, Z. Li, Z. Wu, J. Rhee, and P. Mittal. Towards a Timely Causality Analysis for Enterprise Security. In *Network and Distributed System Security Symposium (NDSS).*

Luo, D., W. Cheng, D. Xu, W. Yu, B. Zong, H. Chen, and X. Zhang (2020). Parameterized explainer for graph neural network.

Malwarebytes (2022). North korea's lazarus apt leverages windows update client, github in latest campaign. https://tinyurl.com/mr4h7d35.

McMahan, B., E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas (2017). Communication-efficient learning of deep networks from decentralized data. In *Artificial Intelligence and Statistics*, pp. 1273–1282. PMLR.

Meidan, Y., M. Bohadana, Y. Mathov, Y. Mirsky, D. Breitenbacher, A. Shabtai, and Y. Elovici (2018). N-baiot: Network-based detection of iot botnet attacks using deep autoencoders. *arXiv preprint arXiv:1805.03409*.

Metasploit (2021). metasploit. https://www.metasploit.com/. (Accessed on 11/29/2021).

MetasploitVenom (2021). Offensive security. https://tinyurl.com/37fdcmkf. Accessed: April 6, 2023.

Michael, N., J. Mink, J. Liu, S. Gaur, W. U. Hassan, and A. Bates (2020). On the forensic validity of approximated audit logs. In *Proceedings of the 36th Annual Computer Security Applications Conference*, pp. 189–202.

Microsoft (2015). Event Tracing. https://docs.microsoft.com/en-us/windows/desktop/ETW/event-tracing-portal.

Milajerdi, S. M., R. Gjomemo, B. Eshete, R. Sekar, and V. N. Venkatakrishnan (2019). HOLMES - Real-Time APT Detection through Correlation of Suspicious Information Flows. In *IEEE Symposium on Security and Privacy (SP).*

mirai (2016). Mirai Attacks. https://goo.gl/QVv89r.

Moosavi-Dezfooli, S.-M., A. Fawzi, and P. Frossard (2016). Deepfool: a simple and accurate method to fool deep neural networks. In *IEEE conference on computer vision and pattern recognition (CVPR).*

Mothukuri, V., P. Khare, R. M. Parizi, S. Pouriyeh, A. Dehghantanha, and G. Srivastava (2021). Federated-learning-based anomaly detection for iot security attacks. *IEEE Internet of Things Journal 9*(4), 2545–2554.

motion (2018). Motion. https://motion-project.github.io/.

Mukherjee, K., J. Wiedemeier, T. Wang, J. Wei, F. Chen, M. Kim, M. Kantarcioglu, and K. Jee (2023). Evading provenance-based ml detectors with adversarial system actions. In *USENIX Security Symposium (SEC)*.

Nguyen, A., J. Yosinski, and J. Clune (2015). Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *IEEE conference on computer vision and pattern recognition (CVPR)*.

Nguyen, T. D., S. Marchal, M. Miettinen, M. H. Dang, N. Asokan, and A.-R. Sadeghi (2018). Diot: A crowdsourced self-learning approach for detecting compromised iot devices. *arXiv preprint arXiv:1804.07474*.

Nõmm, S. and H. Bahşi (2018). Unsupervised anomaly based botnet detection in iot networks. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pp. 1048–1053. IEEE.

NVIDIA (2022). Nvidia jetson nano developer kit — nvidia developer. https://developer.nvidia.com/embedded/jetson-nano-developer-kit. (Accessed on 07/23/2021).

O'Kane, P., S. Sezer, and K. McLaughlin (2011, 05). Obfuscation: The hidden malware. In *IEEE Symposium on Security and Privacy (SP)*.

Ozcelik, M., N. Chalabianloo, and G. Gur (2017). Software-defined edge defense against iot-based ddos. In *2017 IEEE International Conference on Computer and Information Technology (CIT)*, pp. 308–313. IEEE.

Pan, S., R. Hu, G. Long, J. Jiang, L. Yao, and C. Zhang (2019). Adversarially regularized graph autoencoder for graph embedding.

Paramonov, K., D. Shemetov, and J. Sharpnack (2019). Estimating graphlet statistics via lifting. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*.

Phillips, G. (2021). What is a drive-by download malware attack? http://tinyurl.com/yzbecvj5.

Pierazzi, F., F. Pendlebury, J. Cortellazzi, and L. Cavallaro (2020, 05). Intriguing Properties of Adversarial ML Attacks in the Problem Space. In *IEEE Symposium on Security and Privacy (SP)*.

Raza, S., L. Wallgren, and T. Voigt (2013). Svelte: Real-time intrusion detection in the internet of things. *Ad hoc networks 11*(8), 2661–2674.

Redhat (2017). The linux audit framework. https://github.com/linux-audit/.

Rehman, M. U., H. Ahmadi, and W. U. Hassan (2024). FLASH: A Comprehensive Approach to Intrusion Detection via Provenance Graph Representation Learning. In *IEEE Symposium on Security and Privacy (SP).*

Rieger, P., M. Chilese, R. Mohamed, M. Miettinen, H. Fereidooni, and A.-R. Sadeghi (2023, 8). Argus: Context-based detection of stealthy iot infiltration attacks.

Seshadhri, C., A. Pinar, and T. G. Kolda (2013). Fast triangle counting through wedge sampling. In *Proceedings of the SIAM Conference on Data Mining.*

Shahid, O., V. Mothukuri, S. Pouriyeh, R. M. Parizi, and H. Shahriar (2021). Detecting network attacks using federated learning for iot devices. In *2021 IEEE 29th International Conference on Network Protocols (ICNP)*, pp. 1–6. IEEE.

Sikder, A. K., H. Aksu, and A. S. Uluagac (2017, August). 6thSense: A context-aware sensor-based attack detector for smart devices. In *26th USENIX Security Symposium (USENIX Security 17)*, Vancouver, BC, pp. 397–414. USENIX Association.

Sikder, A. K., H. Aksu, and A. S. Uluagac (2020). A context-aware framework for detecting sensor-based threats on smart devices. *IEEE Transactions on Mobile Computing 19*(2), 245–261.

Sikder, A. K., G. Petracca, H. Aksu, T. Jaeger, and A. S. Uluagac (2021). A survey on sensor-based threats and attacks to smart devices and applications.

Sivaraman, V., H. H. Gharakheili, A. Vishwanath, R. Boreli, and O. Mehani (2015). Network-level security and privacy control for smart-home iot devices. In *WiMob*, pp. 163–167.

Someya, M., Y. Otsubo, and A. Otsuka (2023, 02). Fcgat: Interpretable malware classification method using function call graph and attention mechanism. In *Network and Distributed System Security Symposium (NDSS)*, Volume 1.

Song, J., M. Kim, N. D. Lane, R. K. Balan, F. Dang, Z. Li, Y. Liu, E. Zhai, Q. A. Chen, T. Xu, Y. Chen, and J. Yang (2019). Understanding Fileless Attacks on Linux-based IoT Devices with HoneyCloud. *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*.

Song, Y., M. E. Locasto, A. Stavrou, A. D. Keromytis, and S. J. Stolfo (2007). On the infeasibility of modeling polymorphic shellcode. In *ACM conference on Computer and Communications Security (CCS).*

Sood, A. K. and S. Zeadally (2016). Drive-by download attacks: A comparative study. *IT Professional*.

Strike, C. (2023). Cobalt strike — adversary simulation and red team operations. https://www.cobaltstrike.com/.

Tang, Y., D. Li, Z. Li, M. Zhang, K. Jee, X. Xiao, Z. Wu, J. Rhee, F. Xu, and Q. Li (2018, 11). NodeMerge: Template Based Efficient Data Reduction For Big-Data Causality Analysis. In *ACM conference on Computer and Communications Security (CCS)*.

Telychko, V. (2024). Gamaredon attack detection: Cyber-espionage operations against ukraine by the russia-linked apt. https://socprime.com/blog/gamaredon-attack-detection-cyber-espionage-operations-against-ukraine/.

ThreatIntelligence (2023). Mitre att&ck framework: All you ever wanted to know. http://tinyurl.com/5n6jt5pt.

Tramèr, F., A. Kurakin, N. Papernot, I. Goodfellow, D. Boneh, and P. McDaniel (2017). Ensemble adversarial training: Attacks and defenses. *arXiv preprint arXiv:1705.07204*.

trustzone (2018). Introducing Arm TrustZone. https://developer.arm.com/technologies/trustzone.

Ugander, J., L. Backstrom, and J. Kleinberg (2013). Subgraph frequencies: Mapping the empirical and extremal geography of large graph collections. In *Proceedings of the 22nd international conference on World Wide Web*.

Veličković, P., G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio (2017). Graph attention networks. *arXiv preprint arXiv:1710.10903*.

VirusTotal (2021). Virustotal. https://www.virustotal.com/. Accessed: April 6, 2023.

vpnfilter (2018). VPNFilter. https://blog.talosintelligence.com/2018/05/VPNFilter.html.

Wang, J., S. Hao, R. Wen, B. Zhang, L. Zhang, H. Hu, and R. Lu (2020). Iot-praetor: Undesired behaviors detection for iot devices. *IEEE Internet of Things Journal 8*(2), 927–940.

Wang, Q., W. U. Hassan, D. Li, K. Jee, X. Yu, K. Zou, J. Rhee, Z. Chen, W. Cheng, C. A. Gunter, and H. Chen (2020). You Are What You Do: Hunting Stealthy Malware via Data Provenance Analysis. In *Network and Distributed System Security Symposium (NDSS)*.

Warnecke, A., D. Arp, C. Wressnegger, and K. Rieck (2019). Don't paint it black: White-box explanations for deep learning in computer security. *CoRR*.

Warnecke, A., D. Arp, C. Wressnegger, and K. Rieck (2020). Evaluating explanation methods for deep learning in security. In *2020 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*.

Xu, Z., P. Fang, C. L. Liu, X. Xiao, Y. Wen, and D. Meng (2022). DEPCOMM: Graph Summarization on System Audit Logs for Attack Investigation. In *IEEE Symposium on Security and Privacy (SP)*.

Xu, Z., Z. Wu, Z. Li, K. Jee, J. Rhee, X. Xiao, F. Xu, H. Wang, and G. Jiang (2016, 11). High Fidelity Data Reduction for Big Data Security Dependency Analyses. In *ACM conference on Computer and Communications Security (CCS)*.

Yang, F., J. Xu, C. Xiong, Z. Li, and K. Zhang (2023). {PROGRAPHER}: An anomaly detection system based on provenance graph embedding. In *32nd USENIX Security Symposium (SEC)*.

Ying, Z., D. Bourgeois, J. You, M. Zitnik, and J. Leskovec (2019). Gnnexplainer: Generating explanations for graph neural networks. In *Neural Information Processing Systems (NeurIPS)*.

Yuan, H., H. Yu, S. Gui, and S. Ji (2022). Explainability in graph neural networks: A taxonomic survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

Yuan, H., H. Yu, J. Wang, K. Li, and S. Ji (2021). On explainability of graph neural networks via subgraph explorations. In *International Conference on Machine Learning (ICML)*. PMLR.

zeek (2021). Zeek. https://zeek.org/.

Zengy, J., X. Wang, J. Liu, Y. Chen, Z. Liang, T.-S. Chua, and Z. L. Chua (2022). Shadewatcher: Recommendation-guided cyber threat analysis using system audit records. In *IEEE Symposium on Security and Privacy (SP)*.

zigbeeflaw (2015). Critical Flaw identified In ZigBee Smart Home Devices. https://goo.gl/BFBa1X.

BIOGRAPHICAL SKETCH

Kunal Mukherjee was born on September 29, 1997, in Kolkata, India, to Mr. Kingsuk Mukherjee and Dr. Sharmistha Mukherjee. After completing high school at Benjamin Bosse High School in Evansville, Indiana, he pursued a Bachelor's degree in Computer Engineering at the University of Evansville (UE). During his undergraduate years, Kunal developed a deep interest in cybersecurity and intrusion detection, inspired by various computer security and cryptography courses. His passion culminated in his senior thesis, which was selected for presentation at the IEEE MIT Undergraduate Research Technology Conference (URTC) in 2018. While studying at UE, Kunal also interned at Ciholas, Inc. for two and a half years, where he gained hands-on experience with ultra-wideband (UWB) sensor technology and motion sensing technology. In 2019, after completing his Bachelor's degree in three years, he moved to Dallas, Texas to begin a fully-funded PhD program in Computer Science at The University of Texas at Dallas (UTD). Within two years, he completed his qualification exams and became eligible to earn his Master's degree.

His fascination with cybersecurity and intrusion detection led him to work under the guidance of Dr. Kangkook Jee and Dr. Murat Kantarcioğlu. Since then, Kunal has focused on developing lightweight, robust, and explainable intrusion detection systems through system provenance analysis. His contributions include several published papers and a notable internship at Zillow Group, Inc., as an Applied Scientist Intern. Now, as he completes his PhD, Kunal is preparing to embark on a post-doctoral journey.

CURRICULUM VITAE

Kunal Mukherjee

Contact Information:

Department of Computer Science
The University of Texas at Dallas
800 W. Campbell Rd.
Richardson, TX 75080-3021, U.S.A.

Email: `kunmukh@gmail.com`

Educational History:

BS, Computer Engineering, University of Evansville, 2019
MS, Computer Science, The University of Texas at Dallas, 2024
PhD, Computer Science, The University of Texas at Dallas, 2025

IoT Integration, Adversarial Attacks, and Threat Explanations in Provenance-based Intrusion Detection Systems
PhD Dissertation
Computer Science Department, The University of Texas at Dallas
Advisors: Dr. Kangkook Jee and Dr. Murat Kantarcıoğlu

Employment History:

Research/Teaching Assistant, The University of Texas at Dallas, August 2019 – present
Applied Scientist Intern, Zillow Group, Inc., May 2024 – December 2024
Computer Engineering Research Intern, Ciholas, Inc, May 2017 – August 2019

Professional Recognitions and Honors:

Graduated *summa cum laude*, University of Evansville, 2019

Publications:

1. **Kunal Mukherjee**, Joshua Wiedemeier, Tianhao Wang, James Wei, Feng Chen, Muhyun Kim, Murat Kantarcioglu, and Kangkook Jee (2023). "Evading Provenance-Based ML Detectors with Adversarial System Actions." In *Proceedings of 32nd USENIX Security Symposium.*

2. **Kunal Mukherjee**, Joshua Wiedemeier, Tianhao Wang, Muhyun Kim, Feng Chen, Murat Kantarcioglu, and Kangkook Jee (2023). "Interpreting GNN-based IDS detections using provenance graph structural features." In *arXiv.*

3. **Kunal Mukherjee**, Joshua Wiedemeier, Qi Wang, Junpei Kamimura, John Junghwan Rhee, James Wei, Zhichun Li, Xiao Yu, Lu-An Tang, Jiaping Gui and Kangkook Jee (2024). "ProvIoT : Detecting Stealthy Attacks in IoT through Federated Edge-Cloud Security." In *22nd International Conference on Applied Cryptography and Network Security.*

4. Tianhao Wang, Simon Klancher, **Kunal Mukherjee**, Joshua Wiedemeier, Feng Chen Murat Kantarcioglu, Kangkook Jee. (2024)."ProvCreator: Synthesizing Graph Data with Text Attributes." Submitted to *Proceedings of the 13th International Conference on Learning Representations.*

Professional Service:

Reviewer: ACM Computing Surveys, ACM Transactions on Privacy and Security, and ICLR '25
Artifact Evaluation Committee: MobiSys '23, MLSys '23, NDSS '24, and USENIX '24
External Reviewer: ISC '23, CCS '24, and USENIX '24

Honors and Awards:

Winner: Best Poster Presentation - 05/2019
1st Runner Up: Nominee Outstanding Senior Undergraduate Project - 05/2019
Winner: IEEE MIT Undergraduate Research Technology Conference (URTC) - 10/2018
Award: University of Michigan CSE 2018 Workshop Travel Scholarship - 10/2018
Award: University of Evansville Leadership Academy Gold Medal Recipient - 04/2018
Scholarship: University of Evansville International Scholarship - 05/2016
Scholarship: IVY Tech Valedictorian Scholarship - 05/2016
Scholarship: Anna & Benjamin Bosse Scholarship - 05/2016
Scholarship: William N. Lindsey, Sr Scholarship - 05/2016
Scholarship: University of Evansville Mathematics - 04/2016