Evading Provenance-Based ML Detectors with Adversarial System Actions

Kunal Mukherjee, Joshua Wiedemeier, Tianhao Wang, James Wei, Feng Chen, Muhyun Kim, Murat Kantarcioglu, and Kangkook Jee,
The University of Texas at Dallas

https://www.usenix.org/conference/usenixsecurity23/presentation/mukherjee

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Open access to the Proceedings of the 32nd USENIX Security Symposium is sponsored by USENIX.

Evading Provenance-Based ML Detectors with Adversarial System Actions

Kunal Mukherjee, Joshua Wiedemeier, Tianhao Wang, James Wei, Feng Chen, Muhyun Kim, Murat Kantarcioglu, and Kangkook Jee

Department of Computer Science, The University of Texas at Dallas

We present PROVNINJA, a framework designed to generate adversarial attacks that aim to elude provenance-based Machine Learning (ML) security detectors. PROVNINJA is designed to identify and craft adversarial attack vectors that statistically mimic and impersonate system programs.

Leveraging the benign execution profile of system processes commonly observed across a multitude of hosts and networks, our research proposes an efficient and effective method to probe evasive alternatives and devise stealthy attack vectors that are difficult to distinguish from benign system behaviors. PROVNINJA's suggestions for evasive attacks, originally derived in the feature space, are then translated into system actions, leading to the realization of actual evasive attack sequences in the problem space.

When evaluated against State-of-The-Art (SOTA) detector models using two realistic Advanced Persistent Threat (APT) scenarios and a large collection of fileless malware samples, PROVNINJA could generate and realize evasive attack variants, reducing the detection rates by up to 59%. We also assessed PROVNINJA under varying assumptions on adversaries' knowledge and capabilities. While PROVNINJA primarily considers the black-box model, we also explored two contrasting threat models that consider blind and white-box attack scenarios.

1 Introduction

Recent cyber incidents [1]–[4] have demonstrated that conventional security solutions are inadequate against skilled attackers employing stealthy attack vectors. To defend against stealthy attack campaigns, modern security defenses have emerged. Among these, ML-based security defenses has proved effective at processing system [5] and network data [6]. The proliferation of granular system monitoring and efficient system event collection has facilitated ML-based extensions of traditional security solutions based on static artifacts such as file hashes, black-listed domains and IPs, enabling more effective and dynamic security measures.

Low-level system activities represented in *provenance graphs* are used to build ML-based security models, allowing in-depth security monitoring across a high-value confined network [7]–[11]. Provenance graphs causally associate important system interactions among system resources [12], [13] to highlight their control and data dependencies. Provenance-based ML security detectors analyze fine-grained system runtime data to effectively defend against APT campaigns [14]. Despite their effectiveness in protecting modern computing infrastructure, the security and stability of provenance-based ML defenses and ML-assisted applications [15], [16] have not been thoroughly investigated [17]–[19]. This lack of assurance allows for potential vulnerabilities that adversaries can exploit. Threat models for ML-based defenses need to be extended to consider attackers who seek to specifically circumvent these defenses.

In this work, we propose PROVNINJA, a systematic data-driven approach that leverages publicly available program execution profiles [20] to reexamine the current SOTA practices of provenance-based ML security. By modifying attack vectors to avoid conspicuous actions and mimic the normal execution of system programs, PROVNINJA systematically poses difficult challenges to ML-based *Intrusion Detection System* (IDS).

While extensive research has been conducted on the robustness of ML models across numerous application domains by exploiting their instabilities against adversarial samples [16], [21], a substantial gap remains between adversarial attacks in the feature space and their manifestations in the problem space, particularly in domains with strong problem space dependencies [22]. Such discrepancies are especially pronounced in provenance-based ML models, which are constructed using fine-granular datasets gathered from networks of exceptionally complex and opaque systems [23].

In particular, The system provenance domain uniquely presents the following challenges in applying adversarial ML techniques to evade ML detectors: First, the feature space is captured based on the actions in the problem space, distinguishing it from other ML domains such as image processing and textual analysis tasks. In these domains, little transfor-

mative processes are applied to separate the feature space from the problem space before extracting feature embeddings. This approach has significant implications —– any changes in the problem space can result in non-linear changes in the feature space, as it is captured based on problem space actions. Second, publicly available datasets and data collection tools for system provenance are severely limited. To adequately train provenance-based ML detectors, a large and realistic provenance dataset is required. Although public datasets from previous research exist [20], [24], [25], they are designed to support forensic activities and related research rather than data-intensive ML research. Lastly, realizing feature-level attacks in the problem space requires substantial domain knowledge and technical expertise. We encountered several challenges while translating feature space evasions into system actions.

To address above challenges, we propose a systematic approach that (1) locates conspicuous events, (2) modifies those events to evade detection while preserving the adversary's objectives, and (3) realizes the feature-space attacks using concrete system actions for use against real systems.

To validate the coverage and general applicability of our approach, we evaluated ProvNinja against four SOTA ML detectors across two different classes of modeling widely used by previous research: path-based embedding and graph-based embedding. The models are trained with an extensive dataset collected using our in-house deployment which monitored system events on an average of 86 hosts over 13 months. For our malicious dataset, we collected system provenance graphs for two APT scenarios and 5,925 Fileless malware samples [26].

With respect to our assumptions about the adversaries' knowledge and capabilities, we primarily adopt the black-box threat model [27], [28], thereby limiting the adversaries' insight into the victim network's event history. Our research pragmatically employs a surrogate approach, using publicly available datasets [20] to approximate the behavior of system programs commonly persisting across various hosts and networks. To ensure a comprehensive assessment, we also assess the efficacy of our approach under different adversarial assumptions for both blind and white-box models.

In summary, our work brings the following contributions:

- To the best of our knowledge, ProvNinja is the first to systematic study of adversarial evasion of provenance-based ML security detectors using a publicly available surrogate dataset.
- ProvNinja implements a data driven approach to construct evasive attack vectors with minimal human oversight and realistic system constraints.
- We thoroughly evaluate ProvNinja against different ML models using our comprehensive benign and malicious dataset gathered from real-world deployment.

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.

2 System Provenance for Stealthy Attacks

We provide necessary background information on system provenance and provenance-based IDS in the context of stealthy attacks.

2.1 System Provenance

System provenance traces information and control dependencies of a computer system [12], [13]. 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, `execute` and `exit` 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. [29]–[31].

2.2 Provenance-based IDS

Provenance-based IDS have focused on stealthy attacks and APT campaigns to address known limitations of the traditional security defenses. Recent advancements in ML research have extended anomaly detection to provenance graphs [32], 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 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

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

list of structural dependencies among system resources while seen benign to anomaly detection models. Therefore, despite its data collection and modeling costs, the provenance-based IDS 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 [7], [8], [10], [11]. Path-based models extract graph subcomponents (*e.g.,* causal paths) from the provenance graph and vectorize them to leverage existing learning approaches [33], [34]. 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 [35], we can leverage Graph-Neural Network (GNN) [36] techniques to digest the entire provenance graph directly. While promising, GNN-based anomaly detection models are still in its infancy and have not been thoroughly hardened and verified against dedicated adversaries.

In this paper, *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 [8], which uses Local Outlier Factor (LOF) on path embeddings to find outliers; and SIGL [7], 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 S-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 models (§4.2).

2.3 Stealthy Attacks and 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. Interested readers can find more information in §A.2 and §A.3.

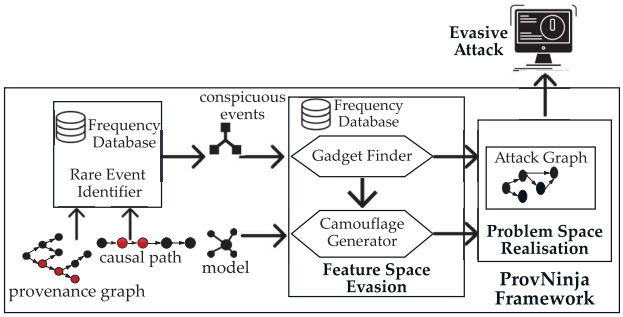


Figure 1: PROVNINJA framework.

3 Problem Statement and Threat Model

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 —— S-GAT and Prov-GAT . As mentioned in §2.2, 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.

3.1 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?* 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 §6, 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 [22].

3.2 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 [8], [10] (§4.1). 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 [37] 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 [22]. We assume a highly skilled and motivated adversary who is capable of devising these stealthy evasive attack vectors.

4 PROVNINJA Overview

There exists a rich literature on adversarial attacks against ML models [22], [38] that affect prediction results with minimal overhead. However, previous exploration of adversarial attacks that can exploit provenance-based threat detectors [23] 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 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 [22].

4.1 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 [7], [8], [10], 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 [8], [10], 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, we use this information to mimic benign process behaviors.

4.2 Provenance-based ML Detectors

To demonstrate generality, we consider four different ML detectors in two modeling categories: *path-based* and *graph-based*, which have different feature representations of provenance graphs. While we are aware of other provenance-based ML detector implementations [11], [39] with different design choices, we believe that our high-level approach will generalize effectively to those ML detectors.

Path-based embedding models. Path-based modeling approaches [7], [8], [10] decompose the provenance graphs into causal paths using random walks. The path components are then annotated with frequency scores using the event histories. These paths, ordered by rarity, are embedded [40] and modeled [33], [34] to reconstruct the context around the Point-of-Interest (PoI) event. While computationally efficient, path-

based approaches lose all structural information outside the path, which can limit their effectiveness against advanced malicious actors.

GNN models. Graph-based detectors directly consume provenance graphs to capture the holistic structure of system activities. GNNs use message passing to summarize and propagate structural features [41], [42]. We consider two different GNN architectures. The first architecture only sees the structure of the graph, and the second architecture sees embeddings of the node attributes in addition to the graph structure. Based on our experiments presented in §6.4, we found that the addition of node attributes results in considerably higher detection power. However, node attributes are vulnerable to manipulation by skilled adversaries, hindering their reliability against APTs.

4.3 Identifying Conspicuous 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 $R_e(u,v,r) = \frac{|Freq(u,v,r)|}{|Freq(u,*,r)|}$ [10]. 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). Interested readers may find the full original attack paths and more comprehensive examples of gadget chains in the appendix in Table 9.

4.4 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 §4.5) that can replace conspicuous events. 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.

Table 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	-

4.5 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 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 $(g_0, \ldots, g_n)$ 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 in §6, we use a constant $T = 0.03$). 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 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. More gadget chains for other components of the APT scenarios can be found in §A.4.

4.7 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 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 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.

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 ≤ 0 then
    return ∅
rare_edges = TopRareEdges(E, K)
gadgets = ⋃_{e∈rare_edges} FindGadgetChains(e, F, T)
foreach g ∈ gadgets do
    G' = ApplyGadget(G, g)
    if M(G') == benign then
        return G'
    G' = ProvNinja(G', F, M, D − 1, T, K)
    if G' ≠ ∅ then
        return G'
return ∅
```

5 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.[22], 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.

5.1 Problem Space Constraints

Pierazzi et al.[22] 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 § 5.2.4, 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 [29]–[31]. Assessing the impact of these data reduction schemes on the effectiveness of evasive attacks renders a promising research direction for future work [43].

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 2. 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 §5.2.

5.2 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 §6.6, we evaluate their effectiveness in reducing the required manual effort.

5.2.1 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 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 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 1 is rejected since `tvnserver.exe` connects externally to manage GPS data. Restarting networking processes, as in gadget chain 17 in Table 1, can indicate APT attacks, trigger security alerts, and lead to system instability or data loss.

5.2.2 Program-Specific Considerations

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 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 1, for example, is automatically rejected because it contains a special sequence.

5.2.3 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 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 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 1, which is also rejected, displays a process attempting to schedule a task, as it requires calling `schtasks.exe`.

5.2.4 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 [44], and proprietary programs like `cargo`, Rust's package manager. Using these programs to create gadget chains, such as 3, 13 in Table 1, leads to unrealizable outcomes in the defender's system due to their unavailability.

6 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 (§6.4) under different threat models (§6.5)?
- **RQ2: Problem Space Attack Realization.** Can ProvNinja's evasive attacks be realized in the problem space (§6.6)?
- **RQ3: Surrogate Data Effectiveness.** How effective is surrogate data in generating evasive attacks? (§6.7)

6.1 Evaluation Methodology

We evaluate ProvNinja's ability to generate evasive attack sequences against provenance-based IDS using our enterprise and supply chain APTs §A.2, and fileless malware §A.3 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 [20] as a surrogate dataset following the blackbox threat model (§3.2). To comprehensively explore ProvNinja's effectiveness on fileless malware, we refer to existing work [26] and collect samples from a popular malware repository [45]. 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 8).

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 S-GAT and Prov-GAT . Additionally, we evaluate ShadeWatcher [32], the SOTA GNN-based anomaly detector for system provenance for the general applicability of ProvNinja approach.

6.2 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 [20] 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

Table 2: 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		Path-based Detectors	Baseline			Random perturbation		PROVNINJA-PATH	
		Precision	Recall	F1	Recall	F1	Recall	F1		Precision	Recall	F1	Recall	F1	Recall	F1
Enterprise APT	S-GAT	0.94	0.74	0.83	0.69 *(-.05)*	0.78 *(-.05)*	0.37 *(-.37)*	0.54 *(-.29)*	ProvDetector	0.98	0.78	0.87	0.88 *(+.10)*	0.92 *(+.05)*	0.23 *(-.55)*	0.31 *(-.56)*
Supply Chain APT		0.93	0.78	0.85	0.96 *(+.18)*	0.91 *(+.06)*	0.44 *(-.34)*	0.53 *(-.32)*		0.99	0.92	0.90	0.95 *(+.03)*	0.95 *(+.05)*	0.35 *(-.57)*	0.30 *(-.60)*
Fileless Malware		0.95	0.94	0.95	0.92 *(-.02)*	0.94 *(-.01)*	0.71 *(-.23)*	0.77 *(-.18)*		0.91	0.91	0.91	0.94 *(+.03)*	0.93 *(+.02)*	0.33 *(-.58)*	0.43 *(-.48)*
Enterprise APT	Prov-GAT	0.95	0.95	0.95	0.71 *(-.24)*	0.68 *(-.27)*	0.25 *(-.70)*	0.37 *(-.58)*	SIGL	0.97	0.99	0.98	0.99 *(+.00)*	0.99 *(+.01)*	0.30 *(-.69)*	0.41 *(-.57)*
Supply Chain APT		0.94	0.96	0.95	0.85 *(-.11)*	0.90 *(-.05)*	0.28 *(-.68)*	0.56 *(-.39)*		0.90	0.90	0.90	0.96 *(+.06)*	0.95 *(+.05)*	0.38 *(-.52)*	0.43 *(-.47)*
Fileless Malware		0.96	0.98	0.97	0.96 *(-.02)*	0.96 *(-.01)*	0.58 *(-.40)*	0.67 *(-.30)*		0.91	0.95	0.93	0.98 *(+.03)*	0.99 *(+.06)*	0.47 *(-.48)*	0.57 *(-.36)*
Average		**0.95**	**0.90**	**0.92**	**0.85** ***(-.05)***	**0.86** ***(-.06)***	**0.44** ***(-.46)***	**0.57** ***(-.35)***		**0.94**	**0.91**	**0.92**	**0.95** ***(+.04)***	**0.96** ***(+.04)***	**0.34** ***(-.57)***	**0.41** ***(-.51)***

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 (§2.3) 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 [26], we collected and ran 5,925 malware samples on our distributed Cuckoo [46] 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 8) 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.[47] and Avllazagaj et al.[48] have emphasized the importance of considering experimental bias in malware analysis. The use of virtual environments like the Cuckoo sandbox [46] 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 [45] 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 [45] 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 [47]; (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 [48]; (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.

6.3 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 2. 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 S-GAT model's average recall and F1 scores across our test cases are 0.82 and 0.88. Notably, the S-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 S-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 S-GAT . ProvDetector is an anomaly detection layer on top of Doc2Vec [40], 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.

6.4 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).

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 2 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 (S-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 conspicuous edges that contributed heavily to the detection of the attack. Against the S-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 2 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.

Table 3: PROVNINJA evasion for ShadeWatcher [32].

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)***

PROVNINJA evasion for ShadeWatcher [32]. In addition to four provenance-based ML detectors, we also conducted a comparison study using ShadeWatcher [32], which implements state-of-the-art anomaly detection for system provenance, extending recommendation systems. The results in Table 3 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

Table 4: 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)*
S-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)***

provenance-based ML detectors.

6.5 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 [49] 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.

We observe in Table 4 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 [23].

6.6 Problem Space Realization

To further refine the feature space of gadgets for implementing problem space evasion attacks, we employ the recommendations from §5.2. Moreover, we assess the amount of effort (in analyst person hours) necessary to execute a gadget chain.

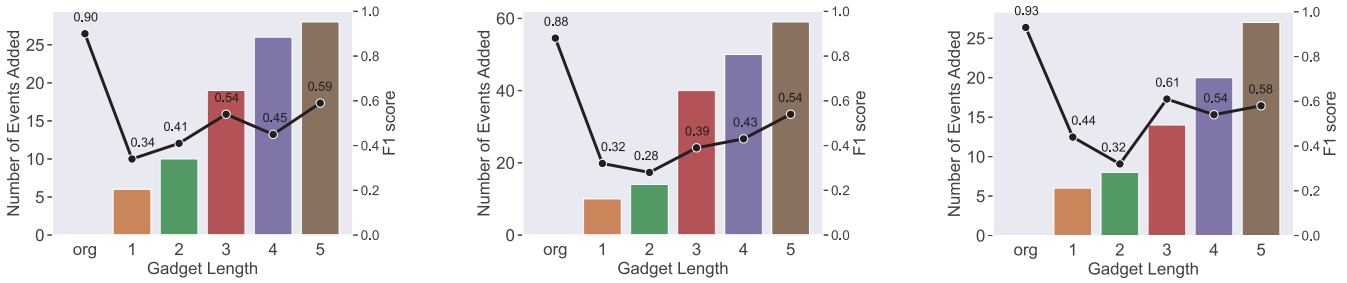
Table 5 summarizes the filtration process of 211 feature space candidates according to the rules outlined in §5.2, resulting in 22 distinct variations for enterprise and supply chain APT scenarios. In the Enterprise APT scenario, 128 initial feature space candidates were reduced by 89% to 14 problem 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.

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 3a reveal that as attack graphs grow larger, implementing evasive attacks becomes increasingly time-consuming.

Interestingly, as shown in Figure 3b, 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 §6.7). This similarity enables the creation of many gadgets using the surrogate dataset on the defender model.

6.7 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.



(a) Enterprise APT. (b) Supply Chain APT. (c) Fileless Malware.

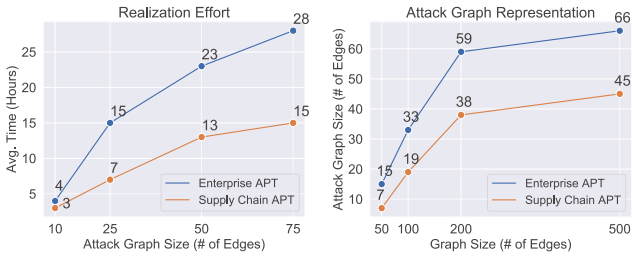
Figure 2: Events added and F1 score vs gadget length. In each evaluation scenario, bars represent number of additional events whereas solid lines are for F-1 score trends.

Table 5: Number of unviable candidates removed by each recommendation §5.2.

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

Table 6: 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		Path-based Detectors	Benign Data		Surrogate Data		Random Data	
		Recall	F1	Recall	F1	Recall	F1		Recall	F1	Recall	F1	Recall	F1
Enterprise APT	S-GAT	0.26	0.35	0.37 *(+.11)*	0.54 *(+.19)*	0.71 *(+.45)*	0.82 *(+.47)*	ProvDetector	0.18	0.15	0.23 *(+.05)*	0.31 *(+.16)*	0.81 *(+.63)*	0.88 *(+.73)*
Supply Chain APT		0.29	0.22	0.44 *(+.15)*	0.53 *(+.31)*	0.96 *(+.67)*	0.91 *(+.69)*		0.25	0.23	0.35 *(+.10)*	0.30 *(+.07)*	0.94 *(+.69)*	0.93 *(+.70)*
Fileless Malware		0.63	0.72	0.71 *(+.08)*	0.77 *(+.05)*	0.93 *(+.30)*	0.94 *(+.22)*		0.29	0.41	0.33 *(+.04)*	0.43 *(+.02)*	0.93 *(+.64)*	0.92 *(+.51)*
Enterprise APT	Prov-GAT	0.17	0.28	0.25 *(+.08)*	0.37 *(+.09)*	0.75 *(+.58)*	0.74 *(+.46)*	SIGL	0.25	0.36	0.30 *(+.05)*	0.41 *(+.05)*	0.99 *(+.74)*	0.99 *(+.63)*
Supply Chain APT		0.21	0.34	0.28 *(+.07)*	0.56 *(+.22)*	0.88 *(+.67)*	0.92 *(+.58)*		0.29	0.38	0.38 *(+.09)*	0.43 *(+.05)*	0.95 *(+.66)*	0.92 *(+.54)*
Fileless Malware		0.55	0.66	0.58 *(+.03)*	0.67 *(+.01)*	0.95 *(+.40)*	0.96 *(+.30)*		0.43	0.51	0.47 *+.04)*	0.57 *(+.06)*	0.97 *(+.54)*	0.95 *(+.44)*
Average		**0.35**	**0.43**	**0.44** ***(+.09)***	**0.57** ***(+.15)***	**0.86** ***(+.51)***	**0.88** ***(+.45)***		**0.32**	**0.39**	**0.39** ***(+.08)***	**0.50** ***(+.11)***	**0.93** ***(+.65)***	**0.93** ***(+.59)***



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

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

Figure 3: 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.

In Table 6, 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.

7 Related Work

Host- and provenance-based IDS. Since Forrest et al. [50] first proposed host-based IDS built on system call trace, a lineage of host-based IDS has been proposed. These host-based IDS take a sequence of system calls without considering arguments, thus raising concerns for their detection accuracy and

stability. Two lines of mimicry attacks were also proposed. Wagner et al. [51] leveraged language automata theory to generate an equivalent sequence of an arbitrary length to evade the prefined rules. Tan et al. [52] exploited the limitations of anomaly detection model by controlling the size of *foreign* sequences. Provenance-based IDS extends sequence-based detectors by taking system provenance graphs as their input. System provenance defines causal dependencies among system resources which are far more comprehensive than system call sequences. So, ML-models built on system provenance graphs are robust against traditional form of mimicry attacks.

Various approaches have been proposed [9], [53]–[56] leveraging system provenance to trace stealthy and long-running APT campaigns. Several heuristics have been proposed to prioritize edges [9] that are likely to involve malicious semantics referring to a threat intelligence [57] source or assigning *tags* [53] that propagate contextual hints to related nodes. Depcomm [56] 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.

Adversarial ML. Adversarial ML research has gained momentum since Kurakin et al. [58] 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 [38], including numerous works [59], [60] aiming to deceive ML models across different domains and modeling approaches.

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.[22] 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, §5 discusses unique challenges of problem space realization in the provenance domain, which is empirically evaluated in §6.6. 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.[23] 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.

8 Discussion and Future Work

Defense against evasive attacks. Although we approach the task from the adversary's perspective, this 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.

System provenance datasets for ML detectors. SOTA ML detectors have focused on proactively identifying stealthy attack campaigns, but have been hampered due to limited provenance datasets. Public datasets [20], [24], [25] support traditional providence research, however these datasets include too few process instances to effectively train ML models. DARPA's popular TC dataset §6.2 includes very detailed, low-level system events (*e.g.,* Windows registry, IPC, Android Intent), but even this dataset is too limited for advanced research, as it only includes records of controlled activities over two weeks. Because of these shortcomings, we have decided to publish our dataset. to facilitate future research in provenance-based ML security.

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.

9 Conclusion

In this paper, we presented PROVNINJA, a data-driven evasive attack generation framework that defeats provenance-based IDS by replicating the behavioral patterns of common system programs. Our research is the first to explore design space of evasive attack generation against provenance-based IDS. Despite significant challenges due to the distance between problem space system actions and their feature space representations, our research successfully generated and actualized evasive attacks that work on real computer systems. We also demonstrated wide coverage and general applicability by evaluating against four ML detectors with different design approaches using extensive datasets.

PROVNINJA leverages a novel awareness of system program execution profiles through public datasets to discover and circumnavigate conspicuous events in an attack vector, then camouflages each process in the attack to mimic benign execution of the impersonated programs. Our findings demonstrate that PROVNINJA has the ability to decrease defense model F1 scores by an average of 46.63%. Furthermore, these evasive actions can be actualized in the problem space, posing a potential threat to provenance-based IDS.

10 Acknowledgements

We thank the anonymous reviewers and our shepherd for their helpful feedback.The research reported herein was supported in part by NSF awards OAC-1828467, OAC-2115094, NIH award 5RM1HG009034-07 and ARO award W911NF-17-1-0356.

References

[1] *Evasive attacker leverages solarwinds supply chain compromises with sunburst backdoor*, https://tinyurl.com/bdz8s5yn, Accessed: April 6, 2023, Dec. 2020.

[2] A. M. Freed, *Inside the darkside ransomware attack on colonial pipeline*, https://tinyurl.com/yrry43sy, Accessed: April 6, 2023, May 2021.

[3] A. Saini and H. Jazi, *North korea's lazarus apt leverages windows update client, github in latest campaign*, https://tinyurl.com/mr4h7d35, Accessed: April 6, 2023, Jan. 2022.

[4] D. Legezo, *Wildpressure targets industrial in the middle east*, https://tinyurl.com/mr2n8hdu, Accessed: April 6, 2023, Nov. 2019.

[5] S. Omar, A. Ngadi, and H. H. Jebur, "Machine learning techniques for anomaly detection: An overview," *International Journal of Computer Applications*, 2013.

[6] Y. Meidan, M. Bohadana, A. Shabtai, *et al.*, "Detection of unauthorized iot devices using machine learning techniques," *arXiv preprint arXiv:1709.04647*, 2017.

[7] X. Han, X. Yu, T. Pasquier, *et al.*, "Sigl: Securing software installations through deep graph learning," in *USENIX Security Symposium (SEC)*, 2021.

[8] Q. Wang, W. U. Hassan, D. Li, *et al.*, "You Are What You Do: Hunting Stealthy Malware via Data Provenance Analysis," in *Network and Distributed System Security Symposium (NDSS)*, Feb. 2020.

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

[10] W. U. Hassan, S. Guo, D. Li, *et al.*, "NoDoze: Combatting Threat Alert Fatigue with Automated Provenance Triage.," in *Network and Distributed System Security Symposium (NDSS)*, Feb. 2019.

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

[12] S. T. King and P. M. Chen, "Backtracking intrusions," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2003.

[13] Y. Liu, M. Zhang, D. Li, *et al.*, "Towards a Timely Causality Analysis for Enterprise Security," in *Network and Distributed System Security Symposium (NDSS)*, Feb. 2018.

[14] *What is xdr?* https://tinyurl.com/mrymbmd5, Accessed: April 6, 2023, 2022.

[15] W. Hardy, L. Chen, S. Hou, Y. Ye, and X. Li, "Dl4md: A deep learning framework for intelligent malware detection," in *Proceedings of the International Conference on Data Science (ICDATA)*, 2016.

[16] B. Kolosnjaji, A. Zarras, G. Webster, and C. Eckert, "Deep learning for classification of malware system call sequences," in *Australasian joint conference on artificial intelligence*, 2016.

[17] M. Sharif, V. Yegneswaran, H. Saidi, P. Porras, and W. Lee, "Eureka: A framework for enabling static malware analysis," in *European Symposium on Research in Computer Security*, 2008.

[18] A.-D. Schmidt, R. Bye, H.-G. Schmidt, *et al.*, "Static analysis of executables for collaborative malware detection on android," in *2009 IEEE International Conference on Communications*, 2009.

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

[20] J. Griffith, D. Kong, A. Caro, *et al.*, "Scalable Transparency Architecture for Research Collaboration (STARC)-DARPA Transparent Computing (TC) Program," Raytheon BBN Technologies Corp. Cambridge United States, Tech. Rep.

[21] S. Gu and L. Rigazio, "Towards deep neural network architectures robust to adversarial examples," *arXiv preprint arXiv:1412.5068*, 2014.

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

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

[24] A. D. Kent, "Comprehensive, multi-source cybersecurity events data set," Los Alamos National Lab.(LANL), Los Alamos, NM (United States), Tech. Rep., 2015.

[25] J. Glasser and B. Lindauer, "Bridging the Gap: A Pragmatic Approach to Generating Insider Threat Data," *IEEE Symposium on Security and Privacy (SP) Workshops*, 2013, (Accessed on 08/09/2022).

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

[27] *Distributed machine learning models done right*, https://www.cybereason.com/blog/distributed-machine-learning-models-done-right, (Accessed on 06/13/2023).

[28] *Introducing ai-powered ioas | crowdstrike*, https://www.crowdstrike.com/blog/introducing-ai-powered-indicators-of-attack-ioas/, (Accessed on 06/13/2023).

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

[30] Y. Tang, D. Li, Z. Li, *et al.*, "NodeMerge: Template Based Efficient Data Reduction For Big-Data Causality Analysis," in *ACM conference on Computer and Communications Security (CCS)*, Nov. 2018.

[31] Z. Xu, Z. Wu, Z. Li, *et al.*, "High Fidelity Data Reduction for Big Data Security Dependency Analyses," in *ACM conference on Computer and Communications Security (CCS)*, Nov. 2016.

[32] J. Zengy, X. Wang, J. Liu, *et al.*, "Shadewatcher: Recommendation-guided cyber threat analysis using system audit records," in *IEEE Symposium on Security and Privacy (SP)*, 2022.

[33] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander, "Lof: Identifying density-based local outliers," 2000.

[34] M. A. Kramer, "Nonlinear principal component analysis using autoassociative neural networks," *AIChE journal*, no. 2, 1991.

[35] *Deep graph library: Easy deep learning on graphs*, https://www.dgl.ai/, (Accessed on 09/21/2021), 2022.

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

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

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

[39] E. Manzoor, S. M. Milajerdi, and L. Akoglu, "Fast memory-efficient anomaly detection in streaming heterogeneous graphs," in *SIGKDD Conference on Knowledge Discovery and Data Mining*, 2016.

[40] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *International conference on machine learning (ICML)*, 2014.

[41] J. You, J. Leskovec, K. He, and S. Xie, "Graph structure of neural networks," in *Proceedings of Machine Learning Research (PMLR)*, Jul. 2022.

[42] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip, "A comprehensive survey on graph neural networks," *IEEE transactions on neural networks and learning systems*, 2020.

[43] M. A. Inam, Y. Chen, A. Goyal, *et al.*, "SoK: History is a Vast Early Warning System: Auditing the Provenance of System Intrusions," in *IEEE Symposium on Security and Privacy (SP)*, May 2023.

[44] *Nssm - the non-sucking service manager alternatives*, https://tinyurl.com/2p8n5kca, Accessed: April 6, 2023, 2021.

[45] *Virustotal*, https://www.virustotal.com/, Accessed: April 6, 2023, 2021.

[46] *Cuckoo sandbox*, https://tinyurl.com/33jdwr93, Accessed: April 6, 2023, 2019.

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

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

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

[50] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff, "A sense of self for Unix processes," in *IEEE Symposium on Security and Privacy (SP)*, 1996.

[51] D. Wagner and P. Soto, "Mimicry attacks on host-based intrusion detection systems," in *ACM conference on Computer and Communications Security (CCS)*, Nov. 2002.

[52] K. M. C. Tan, K. S. Killourhy, and R. A. Maxion, "Undermining an anomaly-based intrusion detection system using common exploits," in *International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2002.

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

[54] M. N. Hossain, S. M. Milajerdi, J. Wang, *et al.*, "Sleuth: Real-time attack scenario reconstruction from cots audit data," in *USENIX Security Symposium (SEC)*, 2018.

[55] P. Fang, P. Gao, C. Liu, *et al.*, "Back-Propagating System Dependency Impact for Attack Investigation," in *USENIX Security Symposium (SEC)*, Aug. 2022.

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

[57] *Mitre att&ck®*, https://attack.mitre.org/, Accessed: April 6, 2023.

[58] A. Kurakin, I. Goodfellow, and S. Bengio, "Adversarial machine learning at scale," *arXiv preprint arXiv:1611.01236*, 2016.

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

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

[61] E. Cozzi, M. Graziano, Y. Fratantonio, and D. Balzarotti, "Understanding linux malware," in *IEEE Symposium on Security and Privacy (SP)*, 2018.

[62] *Initial access: Tactics, techniques, and procedures*, https://attack.mitre.org/tactics/TA0001/, Accessed: April 6, 2023, 2022.

[63] *Compromise client software binary*, https://attack.mitre.org/techniques/T1554/, Accessed on 11/29/2021, 2021.

[64] *Data collection*, https://attack.mitre.org/tactics/TA0004/, Accessed: April 6, 2023, 2021.

[65] *Process injection: Dynamic-link library injection*, https://attack.mitre.org/techniques/T1534/, Accessed: April 6, 2023, 2021.

[66] *Data manipulation*, ATT&CK, Accessed: April 6, 2023, 2021. [Online]. Available: https://attack.mitre.org/tactics/TA0010/.

[67] *Metasploit*, https://www.metasploit.com/, (Accessed on 11/29/2021), 2021.

[68] *Offensive security*, https://tinyurl.com/37fdcmkf, Accessed: April 6, 2023, 2021.

[69] *Cve-2022-21882*, https://tinyurl.com/2p9cmftm, Accessed: April 6, 2023, 2022.

[70] *Gsecdump*, https://attack.mitre.org/software/S0008/, Accessed: April 6, 2023, 2021.

[71] J. Song, M. Kim, N. D. Lane, *et al.*, "Understanding Fileless Attacks on Linux-based IoT Devices with HoneyCloud," *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, 2019.

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

[73] *Fileless threats protection*, https://tinyurl.com/vbb9xk47, Accessed: April 6, 2023, 2020.

A Appendix

A.1 Dataset Statistics

Benign Program Profiles. In this section, we provide detailed statistics on the system provenance graphs used throughout this paper to evaluate ProvNinja. We selected 52 system programs from our event database that are commonly used in APT campaigns from previous studies [7], [8], [13], [61]. The list can be found in Table 7. 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 7). 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 7).

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

Table 7: 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**

(*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 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.

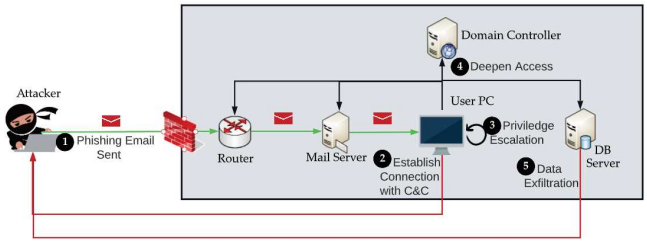


Figure 4: Enterprise APT scenario.

A.2 APT Scenarios

Enterprise APT. The phishing email attack as shown in Figure 4 can be classified according to MITRE ATT&CK framework into five major TTPs: Initial Access [62], Establishing a Foothold [63], Privilege Escalation [64], Deepen Access [65], and Exfiltration [66]. For our experiment, we were able to conduct the five TTPs using the well-known penetration testing framework [67], [68]. The attack involves an attacker crafting a malicious macro (*e.g.,* malware named `java.exe`) embedded attachment (*e.g.,* Excel document) which is sent to a machine victim through email that is inside an enterprise environment, as shown in Figure 4. 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`) [69]. 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 [70] 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.

Supply Chain APT. The supply chain attack Figure 5 also

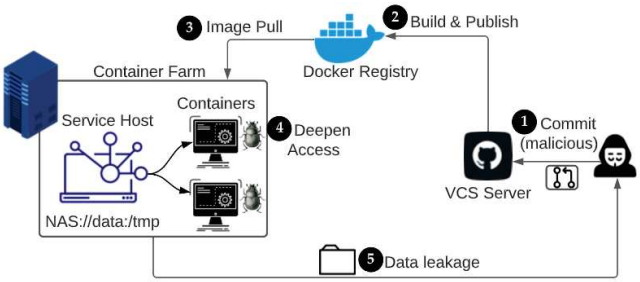


Figure 5: APT attack on Docker supply chain.

Table 8: 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%

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.

A.3 Fileless Malware Evaluation

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[71]–[73]. Referring to the latest research [26], we established a large-scale Fileless malware dataset (refer to Table 8) for our research.

Table 9: 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

A.4 Gadget Chains at Various Stages of APTs

The table Table 9 presented here provides an overview of different gadgets that can be utilized at various stages of enterprise and supply chain APTs. 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.