You Are What You Do: Hunting Stealthy Malware via Data Provenance Analysis

Qi Wang[1], Wajih Ul Hassan[1], Ding Li[2], Kangkook Jee[3], Xiao Yu[2]
Kexuan Zou[1], Junghwan Rhee[2], Zhengzhang Chen[2], Wei Cheng[2], Carl A. Gunter[1], Haifeng Chen[2]
[1]University of Illinois Urbana-Champaign [2]NEC Laboratories America, Inc. [3]University of Texas at Dallas
{qiwang11, whassan3, kzou3, cgunter}@illinois.edu kangkook.jee@utdallas.edu
{dingli, xiao, rhee, zchen, weicheng, haifeng}@nec-labs.com

***Abstract*—To subvert recent advances in perimeter and host security, the attacker community has developed and employed various attack vectors to make a malware much stealthier than before to penetrate the target system and prolong its presence. Advanced malware or "stealthy malware" makes use of various techniques to impersonate or abuse benign applications and legitimate system tools to minimize its footprints in the target system. Thus, it is difficult for traditional detection tools, such as malware scanners, to detect it, as the malware normally does not expose its malicious payload in a file and hides its malicious behaviors among the benign behaviors of the processes.**

In this paper, we present PROVDETECTOR, a provenance-based approach for detecting stealthy malware. Our insight behind the PROVDETECTOR approach is that although stealthy malware attempts to blend into benign processes, the malicious behaviors inevitably interact with the underlying operating system (OS), which will be exposed to and captured by provenance monitoring. Based on this intuition, PROVDETECTOR first employs a novel selection algorithm to identify possible malicious parts in the OS-level provenance data of a process. It then applies a neural embedding and machine learning pipeline to automatically detect any behavior that deviates significantly from normal behaviors. We evaluate our approach on a large provenance dataset from an enterprise network and demonstrate that it achieves very high detection performance of stealthy malware (an average F1 score of 0.974). Further, we conduct thorough interpretability studies to understand the internals of the learned machine learning models.

I. INTRODUCTION

The long-lasting arms race on security warfare has entered a new stage. Malware detection has greatly advanced beyond traditional defenses [1], [2] due to innovations such as machine learning based detection [3], [4], [5], [6] and threat intelligence computing [7]. However, the attacker community has also sought for sophisticated attack vectors to keep up with the advances. Adversaries are now increasingly focusing on new techniques to evade detection and prolong their presence on the target system.

A new kind of technique, *i.e.,* stealthy malware, hides the malware's (or an attacker's) identity by impersonating well-trusted benign processes. Besides simple methods such as renaming processes and program file names, more advanced stealthy techniques are being actively developed and employed. Unlike the traditional malware family that persists on the disk for its payload, stealthy malware hides its malicious logic in the memory space of well-trusted processes, or stores it into less attended locations such as Windows registry or service configurations. Recent reports [8] have estimated that stealthy malware constituing 35% of all attacks, grew by 364% in the first half of 2019, and these attacks are are ten times more likely to succeed copmared to traditional attacks [9], [10].

Despite the importance and urgency of stealthy malware, we are yet to see any definitive solution that detect stealthy malware that employs advanced impersonation techniques. One reason is that stealthy malware minimizes the usage of regular file systems and, instead, only uses locations of network buffer, registry, and service configurations to evade traditional file-based malware scanners. To make things worse, the attacker has multiple options to craft new attacks as needed using different impersonation techniques. First, the attack can take advantage of the well-trusted and powerful system utilities. The latest OSes are shipped with well-trusted administrative tools to ease the system operations, but hese tools are commonly abused targets. For instance, PowerShell and WMIC Windows Management Instrumental Command-line (WMIC)) have long histories of being abused by attackers [11]. Second, an attack can inject malicious logic into benign processes via legitimate OS APIs (*e.g., CreateRemoteThread()* of Win32 API) or use shared system resources. Finally, the attack can exploit vulnerabilities of a benign program to gain control. Since attackers have so many options, the detection approaches that are based on static or behavioral signatures cannot keep up with the evolution of stealthy malware.

Based on the characteristics of stealthy malware, we suggest that an effective defense needs to meet the following three principles. First, the defense technique should not be based on static file-level indicators since they are not distinguishable for stealthy malware. Second, the technique should be able to detect abnormal behavior of well-trusted programs as they are susceptible to attackers with stealthy attack vectors. Third, the technique should be light-weight so as to capture each target program's behavior at a detailed level from each host without deteriorating usability.

Kernel-level (*i.e.,* OS-level) provenance analysis [12], [13], [14], [15], [16] is a practical solution that is widely adopted in real-world enterprises to pervasively monitor and protect their systems. Even when malware could hijack a benign

Network and Distributed Systems Security (NDSS) Symposium 2020
23-26 February 2020, San Diego, CA, USA
ISBN 1-891562-61-4
https://dx.doi.org/10.14722/ndss.2020.24167
www.ndss-symposium.org

process with its malicious logic, it still leaves traces in the provenance data. For example, when a compromised benign process accesses a sensitive file, the kernel-level provenance will record the file access activity. OS kernel supports data collection for provenance analysis incurring only a reasonable amount of overhead when it is compared to heavy-weight dynamic analyses such as virtual machine (VM) assisted-instrumentation or sandbox execution [17], [18].

We propose PROVDETECTOR, a security system that aims to detect stealthy impersonation malware. PROVDETECTOR relies on kernel-level provenance monitoring to capture the dynamic behaviors of each target process. PROVDETECTOR then embeds provenance data to build models for anomaly detection, which detect a program's runtime behaviors that deviate from previously observed benign execution history. Thus it can detect previously unseen attacks. To hunt for stealthy malware, PROVDETECTOR employs a neural embedding model [19] to project the different components in the provenance graph of a process into a n-dimensional numerical vectors space, where similar components are geographically closer. Then a density-based novelty detection [20] method is deployed to detect the abnormal causal paths in the provenance graph. Both the embedding model and the novelty detection model are trained with only benign data. However, while the design insight of PROVDETECTOR to capture and build each program's behavioral model using provenance data seems plausible, the following two challenges must be addressed to realize PROVDETECTOR.

C1: Detection of marginal deviation. Impersonation-based stealthy malware tends to incur only marginal deviation for its malicious behavior, so it can blend into a benign program's normal behavior. For instance, some stealthy malware only creates another thread to plant its malicious logic into the victim process. The victim process still carries out its original tasks, but the injected malicious logic also runs alongside it. Therefore, PROVDETECTOR needs to accurately identify and isolate the marginal outlier events that deviate significantly from the program's benign behaviors. Conventional model learning is likely to disregard such a small portion of behavior as negligible background noise, resulting in misclassification of malicious behaviors.

To address the first challenge, PROVDETECTOR breaks provenance graphs into causal paths and uses the causal paths as the basic components for detection (§V-C).The insight of this decision is that the actions of stealthy malware have logical connections and causal dependencies [21], [**?**], [14]. By using causal paths as detection components, PROVDETECTOR can isolate the benign part of the provenance graph from the malicious part.

C2: Scalable model building. The size of the provenance graph grows rapidly over time connecting an enormous number of system objects. For a provenance-based approach which takes provenance data as its input and builds a model for each process, it is common to see that even in a small organization that has over hundreds of hosts, the system events reported from each end-host incur significant data processing pressure. While simplistic modeling [22] is based on a single-hop relation scale to digest large-scale provenance graphs, the single-hop relation cannot capture and embed contextual causality into the model. However, a modeling that is based on a multi-hop relation (*e.g.,* n-gram [23] or sub-graph matching [24]) would incur huge computation and storage pressure, making it infeasible for any realistic deployment.

To address this second challenge, PROVDETECTOR only processes the suspicious part of a provenance graph. This is achieved by a novel path selection algorithm (§V-C1) that only selects the top K most uncommon causal paths in a provenance graph. Our insight is that the part of a provenance graph that is shared by most instances of a program is not likely to be malicious. Thus, we only need to focus on the part that is uncommon in other instances. Leveraging this path selection algorithm, PROVDETECTOR can reduce most of the training and detection workload.

To confirm the effectiveness of our approach, we conducted a systematic evaluation of PROVDETECTOR in an enterprise environment with 306 hosts for three months. We collected benign provenance data of 23 target programs and used PROVDETECTOR to build their detection models. We then evaluated them with 1150 stealthy impersonation attacks and 1150 benign program instances (50 for each target program). PROVDETECTOR achieved a very high detection performance with an average F1 score of 0.974. We also conducted systematic measurements to identify features contributing to PROVDETECTOR's detection capability on stealthy malware. Our evaluation demonstrated that PROVDETECTOR is efficient enough to be used in a realistic enterprise environment.

To summarize, in this paper, we make the following contributions:

- We designed and implemented PROVDETECTOR, a provenance-based system to detect stealthy malware that employs impersonation techniques.
- To guarantee a high detection accuracy and efficiency, we proposed a novel path selection algorithm to identify the potentially malicious part in the provenance graph of a process.
- We designed a novel neural embedding and machine learning pipeline that automatically builds a behavioral profile for each program and identifies anomalous processes.
- We performed a systematic evaluation with real malware to demonstrate the effectiveness of PROVDETECTOR. We further explained its effectiveness through several interpretability studies.

II. BACKGROUND AND MOTIVATION

In this section, we introduce the stealthy malware we focus on in this study and present our insights of using provenance analysis to detect such malware.

A. Living Off the Land and Stealthy Attacks

"Living off the land" has been a popular cyberattack trend over the last few years. It is characterized by the usage of trusted off-the-shelf applications and preinstalled system tools to conduct stealthy attacks. Since many of these tools are used by system administrators for legitimate purposes, it is harder for the defenders to completely block access to these tools for attack prevention.

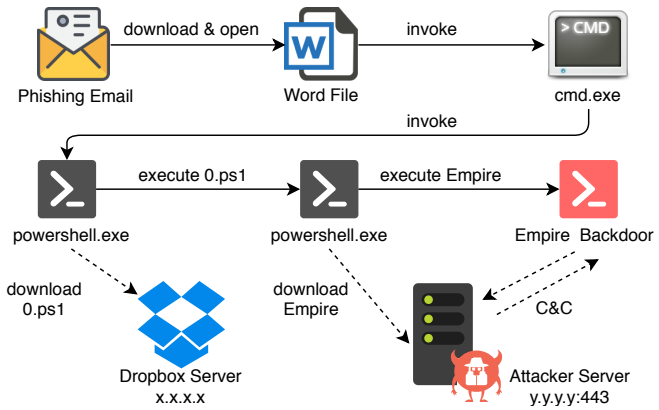


Fig. 1: The kill chain of the DDE script-based attack [30].

Stealthy impersonation malware, which has been increasingly employed in recent cyberattacks [25], [8], heavily uses the "living off the land" strategy to try to evade detection. Instead of storing its payload directly onto a disk and executing it, the malicious code is typically injected into some running processes (often trusted applications or system tools) and executed only within the process memory (*i.e.,* RAM). There are multiple ways to achieve such impersonation purpose.

Memory Code Injection. Memory code injection allows the malware to inject malicious code into a legitimate process' memory. These attacks often targets long-running, trusted system processes (*e.g.,* `svchost.exe`) or applications with valuable user information (*e.g.,* Web Browser). Some well-known code injection techniques include remote thread injection, reflective DLL injection [26], portable executable injection, and recently discovered process hollowing [27] and shim-based DLL injection [28].

Script-based Attacks. Attackers can embed scripts in benign documents like Microsoft Office documents to run their malicious payload. Worse, the Windows system opens access to its core functionalities via various language interfaces (*e.g.,* PowerShell, .Net) that an attacker could take advantage of. Such dynamic languages facilitate execution of a malicious logic on-the-fly, leaving little or no footprints on the filesystem.

Vulnerability Exploits. The third way is to take advantages of the vulnerabilities of a benign software. For example, CVE-2019-0541 [29] allows adversaries to execute arbitrary code in Internet Explorer (IE) through a specially crafted web page.

In Figure 1, we show the kill chain of a real-world DDE[1] (Dynamic Data Exchange) script-based attack, which launches several stages of PowerShell scripts in memory, reported by the Juniper Threat Labs [30]. The attack starts from an email phishing campaign which includes a seemly benign Microsoft Word (MS Word) document as an attachment. When a user opens the document, a message box is shown to enable DDE. Once the DDE is enabled, the embedded `DDEAUTO` command invokes `cmd.exe`, which executes `powershell.exe` to download and execute a PowerShell script (`0.ps1`) using Dropbox service. The `0.ps1` script then introduces the next PowerShell module called "Empire" [31] to open encrypted backdoor. Note that both of the downloaded PowerShell scripts are obfuscated and resided only in memory.

[1]The Dynamic Data Exchange (DDE) is a protocol of Microsoft Windows for sharing data between applications.

B. Existing Detection Methods for Stealthy Malware

Existing detection methods, such as anti-virus (AV) software, use a combination of the following practices [32] to detect malware. As we will discuss, these methods are ineffective at detecting stealthy malware.

Memory Scanning. AV software offers memory scanning as one of their multi-layered solutions. Such techniques scan memory just-in-time at the loading point or in a scheduled way. However, this approach essentially is looking for known payloads in memory. Adversaries can customize or obfuscate the attack payload to avoid detection.

Lockdown Approaches. Lockdown approaches, such as application control or whitelisting, do not help much as stealthy malware often leverages administrative tools or other applications that are typically in a company's whitelist of trusted applications. The defenders could not completely block access to these programs to block the attacks.

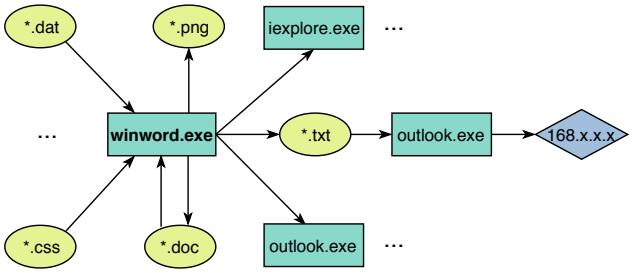
Email Security and Network Security. As shown in Figure 1, script-based malware is often spread through phishing emails. Many security vendors provide solutions for email and network security by inspecting and blocking suspicious attacks by evaluating URLs, attachment files, and scripts. However, similar to the limitation of memory scanning, attack payload is easy to be modified to avoid detection.

In particular, the existing in-host defenses are effective against known file-based malware families. However, the characteristics of stealthy malware, such as low attack footprint, absence of files, usage of dual-use tool, make the detection difficult for existing methods.

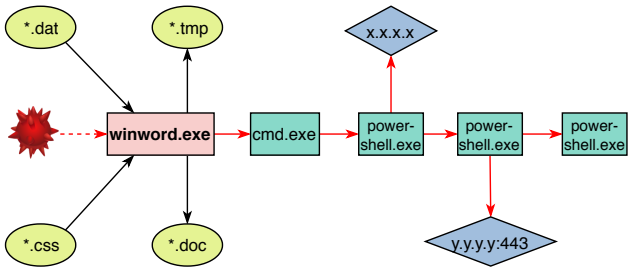
C. Detecting Stealthy Malware Using Provenance Analysis

As discussed in §II-B, existing methods are ineffective at detecting stealthy malware. Since it has multiple characteristics to evade detection, we propose to detect stealthy malware by inspecting its behavior. More specifically, our approach tracks and analyzes the system provenance data related to a program to hunt down stealthy attacks based on behavior differences.

Figure 2 illustrates an example of a stealthy attack and the provenance graphs of two process instances of MS Word (`winword.exe`) with and without an attack. In Figure 2a, we show the provenance graph of a benign instance of MS Word. A benign MS Word process typically reads multiple types of files (*e.g.,* dat, doc, css) created by other programs or itself and writes new files (*e.g.,* doc, txt, png). The created files will also be read by other programs like the Outlook email client (*e.g.,* sent as an attachment). It can also start other programs such as Internet Explorer (`iexplore.exe`) when a user clicks the URLs in a doc file. In contrast, Figure 2b shows the provenance graph of a malicious instance of MS Word, which is used in the DDE script-based attack as shown in Figure 1. Note that we highlight the key attack paths with red arrows. Similar with the benign instance, this malicious MS Word instance also reads and writes different types of



(a) The provenance graph of a benign instance of MS Word.



(b) The provenance graph of a malicious instance of MS Word.

Fig. 2: An illustration of the behavior differences of a benign process instance and a malicious process instance of MS Word (`winword.exe`) using provenance graphs.

files. However, it starts a `cmd.exe` process, which further spawns several `powershell.exe` processes. This behavior is very different from that of the benign one.

Once these process behaviors are represented as provenance graphs, these attack paths become very distinguishable from benign ones. Therefore, provenance tracking and analysis is a key technique to detect stealthy attacks. On the other hand, as shown in Figure 2b, since stealthy attacks take advantages of processes already running in the system, their malicious behaviors could be hidden in benign behaviors of the processes. Moreover, to make the attacks stealthy, malware could mimic and blend in existing benign behaviors. Thus, it is a main challenge to accurately capture the robust and stable features from provenance graphs that can effectively differentiate malicious behaviors from benign ones.

III. Threat Model and Assumptions

In this paper, we focus on *stealthy malware* (or *stealthy attack*) that impersonates or abuses legitimate tools or services already present on the victim's host or exploits trusted off-the-shell applications (*e.g.,* applications in the whitelist of an enterprise's intrusion detection system) to perform malicious activities. As discussed in §II-A, such attacks could conduct extremely damaging activities such as exfiltrating sensitive data, crippling computers, or allowing remote access. Exploiting legitimate tools or applications enable those attacks to do their malicious activities while blending in with normal system behavior and leaving fewer footprints, which makes their detection very difficult. Such stealthy attacks can be achieved through:

- Impersonation techniques such as memory code injection, script-based attacks and vulnerability exploits as described in §II-A.

TABLE I: The system entities and their relations we consider.

Src Entity	Dst Entity	Attributes	Relations
Process	Process	Executable path, Pid, Host name	Start, End
	File	File path, Host name	Read, Write, Execute
	Socket	Src IP, Src port, Dst IP, Dst port	Read, Write

- A malicious version of a trusted application accidentally installed by the user with attack payloads embedded.

Traditional malware that needs to drop a custom built malware binary to the victim's machine to execute its payload is out of our scope. We make the following assumptions about our system. Similar with existing provenance-based systems [14], [15], [13], [33], [34], [16], we assume the underlying OS and the provenance tracker are in our trusted computing base (TCB). We assume the attacker cannot manipulate or delete the provenance record, *i.e.,* log integrity is maintained at all time. Detecting violations of log integrity is a well-studied and orthogonal problem [35]. We also do not consider the attacks performed using implicit flows (side channels) that bypass the system call interface and thus cannot be captured by the underlying provenance tracker. Finally, since our system tries to differentiate benign process instances from malicious ones, we assume that our system has benign provenance data for each monitored program to profile its normal behaviors.

IV. Problem Definition and Basic Assumptions

As discussed in §II-C, the key to detect stealthy malware is to examine the behavior of a process. We thus propose to detect stealthy malware via provenance analysis. In this section, we formally define several concepts that will be used in the rest of this paper and then we formulate the problem statement for ProvDetector.

A. Definitions

System Entity and System Event. Similar with [14], [22], [36], we consider the following three types of system entities: processes, files and network connections (*i.e.,* sockets). A system event $e = (src, dst, rel, time)$ models the interaction between two system entities, where src is the source entity, dst is the destination entity, rel is the relation between them (*e.g.,* a process writes a file), and $time$ is the timestamp when the event happened. Note that, only the process entity can be the source entity in a system event. Each system entity is associated with a set of attributes. For example, a process entity has attributes like its pid and the executable path. In Table I, we show the entity attributes and relations we consider.

System Provenance Graph. Given a process p (identified by its process id and host) in the system, the system provenance graph (or dependency graph) of p is the graph that contains all the system entities that have control dependencies (*i.e.,* start or end) or data dependencies (*i.e.,* read or write) to p. Formally, the provenance graph of p is defined as $G(p) =< V, E >$, where V and E are the sets of vertexes and edges respectively. Vertexes V are system entities and edges E are system events.

Process Instance. We refer a program (or an application) we are interested in monitoring as a program. For example, some

trusted applications like MS Word. A process is an execution of a program. A *process instance* of a program is the process created in one execution of the program.

B. Problem Statement

Suppose we have a set of n provenance graphs $s = \{G_1, G_2, \ldots, G_n\}$ for n benign process instances of a program A. Given a new process instance p of A, we aim to detect if its provenance graph $G(p)$ is benign or malicious. Here and hereafter, we refer to a malicious process instance of A as the process hijacked or abused by a stealthy malware. The provenance graph of the malicious process is thus referred to as *a malicious provenance graph*.

V. ProvDetector

In this section, we detail the design and implementation of ProvDetector.

A. Overview

To detect stealthy malware, we make the following design decisions about ProvDetector:

- ProvDetector is an anomaly detection based technique that only learns from *benign* data.
- ProvDetector uses *causal paths*, *i.e.,* ordered sequences of system events with causal dependency, in provenance graphs as features for detection.
- ProvDetector only learns a *subset* of causal paths of a provenance graph.

We design ProvDetector as an anomaly detection based technique [37] for two reasons: first, it is able to detect unknown attacks (as well as zero-day attacks) as it models the normal operation of a system; second, as the normal profiles are tailored for every application or system, it is very difficult for an attacker to know what activities he can carry out to evade detection. ProvDetector uses causal paths as features to distinguish the malicious part of the provenance data from the benign part. As shown in §VI, this decision helps ProvDetector improve the detection performance. ProvDetector selects a subset of causal paths from a provenance graph to address the dependency explosion problem [38], [33] and to accelerate the speed of both training and detection.

In Figure 3, we show the workflow of ProvDetector which comprises four stages: graph building, representation extraction, embedding, and anomaly detection. ProvDetector is configured to monitor a list of M programs (*e.g.,* Microsoft Word or Internet Explorer) and detect if they are hijacked by stealthy malware. To do this, ProvDetector deploys a monitoring agent on each monitored host, collects system provenance data as we defined in §IV-A, and stores the data in a centralized database. ProvDetector's data collection follows the same principles as previous work [14], [15]. Then, ProvDetector periodically scans the database and checks if any of the newly added processes has been hijacked. For each given process, ProvDetector first builds its provenance graph (Stage: Graph Building). Then it selects a subset of paths from the provenance graph (Stage: Representation Extraction) and converts the paths into numerical vectors (Stage: Embedding). After that, ProvDetector uses a novelty/outlier detector to get predictions for the embedding vectors and reports its final decision (*i.e.*, if the process has been hijacked) (Stage: Anomaly Detection).

ProvDetector has two modes: the training mode and the detection mode. The workflow of the detection mode is described above. The workflow of the training mode is similar. The only difference is that instead of querying the novelty/outlier detector, ProvDetector uses the embedding vectors to train the detector (*i.e.,* building the normal profiles for the applications). Next, we present each stage in detail.

B. Provenance Graph Building

Given a process instance p (identified by its process id and host), ProvDetector builds its provenance graph $G(p) =< V, E >$ as a labeled temporal graph using the data stored in the database. As defined in §IV-A, the nodes V are system entities whose labels are their attributes, and E are edges whose labels are relations and timestamps. Each node in V belongs to one of the following three types: processes, files or sockets. We define each edge e in E as $e = \{src, dst, rel, time\}$. The construction of a provenance graph $G(p)$ starts from $v == p$. Then we add any edge e and its source node src and destination node dst to the graph if $e.src \in V$ or $e.dst \in V$.

C. Representation Extraction

After the provenance graph is built, the next step is representation extraction, the goal of which is to find representations (or features) from the graph to differentiate benign ones and malicious ones. One naive approach is to use the provenance graph itself as the representation. However, as the discussions in §II-C and §VI-C2, the whole provenance graph is not a good representation for detecting stealthy malware as the majority parts of the graph are still benign (the attacks try to blend their attack activities with the normal activities to evade detection).

To isolate the malicious parts from the whole provenance graph, we propose to select certain *causal paths* as the features for the graph. In Figure 4, we show some causal paths from the provenance graphs in Figure 2. Formally, we define a causal path λ in a dependency graph $G(p)$ as an ordered sequence of system events (or edges) $\{e_1, e_2, \ldots, e_n\}$ in $G(p)$, where $\forall e_i, e_{i+1} \in \lambda$, $e_i.dst == e_{i+1}.src$ and $e_i.time < e_{i+1}.time$. Note that the time constraint is important since an event can only be depended on events in the future. Due to the time constraints, ProvDetector will not generate infinite number of paths in loops. For each selected path, ProvDetector removes the host-specific or entity-specific features, such as host name and process identification (PID), from each node and edge. This process ensures that the extracted representation is general for the subsequent learning tasks.

1) Rareness-based Path Selection: Directly extracting all paths from a provenance graph may cause the "dependency explosion problem" [15]. The number of paths is exponential to the number of nodes. Since a provenance graph may contain thousands of nodes [15], it is impossible to traverse all its paths. To address this problem, we propose a rareness-based path selection method that only selects the K most uncommon paths from a provenance graph.

Our intuition is as follows. A process instance of a program may contain two types of workloads: the universal workload

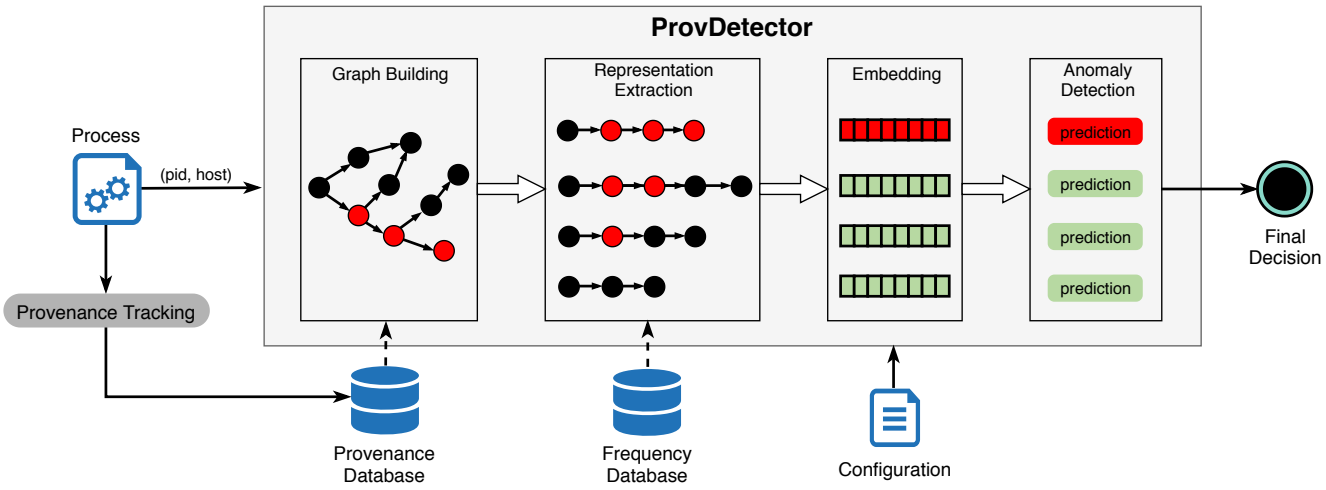


Fig. 3: The overview of PROVDETECTOR

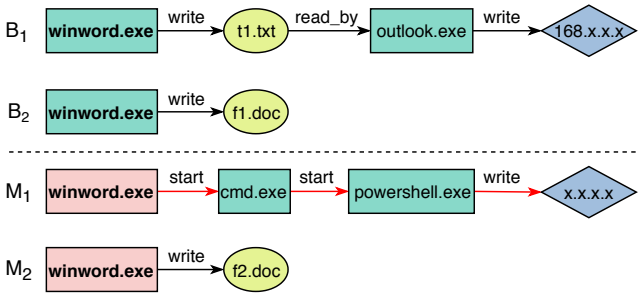


Fig. 4: Example causal paths from the provenance graphs in Figure 2. We concretize the * with file names.

and the instance-specific workload. The universal workloads are common across all instances of the same program and are thus less likely to be malicious. For example, the MS Word program loads a fixed set of DLL files required during its initial stage. This workload is universal to all its instances. On the other hand, the instance-specific workloads, which are different from instance to instance based on the inputs. We argue that malicious workloads are more likely to be instance-specific.

Therefore, we propose to select causal paths that are generated by the instance-specific workloads instead of those paths generated by universal workloads. We determine whether a path is generated from universal workloads or instance-specific workloads by its rareness: more rare a path is, more likely it is from the instance-specific workload.

To discover the top K rarest paths, we use the *regularity score* proposed in previous work [14]. The *regularity score* of a path $\lambda = \{e_1, e_2, \ldots, e_n\}$ is defined as $R(\lambda) = \prod_{i=1}^{n} R(e_i)$, where $R(e_i)$ is the *regularity score* of event e_i. In PROVDETECTOR, the *regularity score* of an event $e = \{src \rightarrow dst\}$ is defined as:

$$R(e) = OUT(src)\frac{|H(e)|}{|H|}IN(dst) \quad (1)$$

In Equation 1, $H(e)$ is the set of hosts that event e happens on while H is the set of all the hosts in the enterprise [15], [14]. To calculate IN and OUT for a node v, PROVDETECTOR partitions the training data into n time windows $T = \{t_1, t_2, \ldots, t_n\}$. We say t_i is *in-stable* if no new in edges are added to v during t_i. Similarly, t_i is *out-stable* if no new out edges are added to v during t_i. Then the $IN(v)$ and $OUT(v)$ are calculated using Equation 2 and Equation 3 respectively where $|T'_{from}|$ is the count of stable windows in which no edge connects from v, $|T'_{to}|$ is the count of stable windows in which no edge connects to v, and $|T|$ is the total number of windows.

$$IN(v) = \frac{|T'_{to}|}{|T|} \quad (2) \qquad OUT(v) = \frac{|T'_{from}|}{|T|} \quad (3)$$

By defining the regularity score, we formalize our path selection problem as *finding the top K paths with the lowest regularity scores from a provenance graph.* To efficiently solve this problem, PROVDETECTOR further converts it to a K longest path problem [39]. To do this, for a provenance graph G, we add a pseudo source node v_{source} to all the nodes whose in-degree are zero and a pseudo sink node v_{sink} to all the nodes whose out-degree are zero. This process converts G to a single source and single sink flow graph G'. We then assign a distance to each edge e as $W(e) = -\log_2 R(e)$ (the outgoing edges of v_{source} and incoming edges of v_{sink} are all uniformly initialized to 1). Thus, the length of λ could be converted as $L(\lambda) = \sum_{i=1}^{n} W(e_i) = -\log_2 \prod_{i=1}^{n} R(e_i)$. Hence, the K longest paths in G' are the K paths with lowest regularity scores in G.

Although solving the K longest path problem on a general graph is an NP-hard problem, it could be efficiently solved by reducing it to the K longest paths problem on a Directed Acyclic Graph (DAG), which can be efficiently solved by the Epstein's algorithm [40] with a time complexity linear to the number of nodes. To reduce our problem to the K longest paths problem on a DAG, we convert G' to a DAG. For each node N in G', PROVDETECTOR orders all its in-edges and out-edges in the temporal order. Then N is split into a set of nodes $\{n_1, n_2, n_3, \ldots, n_i\}$. Any n_i has the same attributes as N but guarantees that *all its in-edges are temporally earlier than any of its out-edges*. As PROVDETECTOR requires all events on a causal graph are temporally ordered, splitting a node based on the temporal orders of its in-edges and out-

edges removes all loops in the graph. After the conversion, PROVDETECTOR relies on existing algorithm [40] to find the K longest paths on the DAG.

D. Embedding

After we select the top K rarest paths as features, the next question is how to feed the paths to anomaly detection models. There are several challenges: (1) the lengths of causal paths are different, and (2) the labels of nodes and edges are unstructured data such as file names or executable paths.

Intuition With the background on word and document embeddings presented in §A, an important intuition we have is to view a causal path as a sentence/document: *the nodes and edges in the path are words that compose the "sentence" which describes a program behavior*. In other words, different nodes and edges compose paths in a similar way that different words compose sentences. Based on this intuition, we could treat each node as a "noun", treat each edge as a "verb", and use their labels to form a sentence that represents the path. For example, for the path B_1 in Figure 4, it can be directly mapped to the following sentence: *Process:winword.exe write File:t1.txt read_by Process:outlook.exe write Socket:168.x.x.x*.

Embeddings Learning To learn an embedding vector for a causal path, we can leverage the document embeddings model with the path as a sentence. Formally, a causal path λ can be translated to a sequence of words $\{l(e_i.src), l(e_i), l(e_i.dst), \ldots, l(e_n.src), l(e_n), l(e_n.dst)\}$, where l is a function to get the text representation of a node or an edge. Currently, we represent a process node by its executable path, a file node by its file path, and a socket node by its source or destination IP and port; we represent an edge by its relation.

With the translated sentences, PROVDETECTOR uses the PV-DM model (explained in Appendix A) of doc2vec [19] to learn the embedding of paths. This method has several advantages. First, it is a self-supervised method, which means we can learn the encoder with purely benign data. Second, it projects the paths to the numerical vector space so that similar paths are closer (*e.g.,* B_2 and M_2 in Figure 4) while different paths are far away (*e.g.,* B_1 and M_1 in Figure 4). This allows us to apply other distance based novelty detection methods in the next step. Third, it also considers the order of words, which is also important. For example, a `cmd.exe` starting a `winword.exe` is likely benign while a `winword.exe` starting a `cmd.exe` is often malicious.

E. Anomaly Detection

The final step of PROVDETECTOR is to use a novelty detection method to detect if the embedding of a path is abnormal. Our design of the anomaly detector is based on the nature of the provenance data. In our observation, provenance data has two important features. First, they cannot be modeled by a single probability distribution model. Modern computer systems and programs are complex and dynamic, it is very hard to model the behaviors of programs with a mathematical distribution model. Second, provenance data have multiple clusters. Workloads of a program can be very different. Although provenance data from similar workloads may look similar, they will be very different if they are from two distinct workloads. Thus, it is very hard to use a single curve to separate normal and abnormal provenance data in the embedding space.

Based on the features of provenance data, PROVDETECTOR uses Local Outlier Factor (LOF) [20] as the novelty detection model. LOF is a density based method. A point is considered as an outlier if it has lower local density than its neighbors. LOF does not make any assumption on the probability distribution of data nor separates the data with a single curve. Thus, it is an appropriate method for our novelty detection problem.

Final Decision Making In the detection phase, we use the built novelty detection model to make predictions of path embedding vectors of a provenance graph. We then use a threshold-based method, *i.e.,* if more than t embedding vectors are predicted as malicious we treat the provenance graph as malicious, to make the final decision about whether the provenance graph is benign or malicious. This method could enable an early stop in the path selection process to reduce detection overhead when the top t instead of K selected paths are already predicted as malicious.

F. Implementation

While PROVDETECTOR takes inputs from both Linux and Windows hosts, our evaluation focuses on Windows event, as our benign deployment mainly comprise of Windows host and most stealthy malware runs for Windows target. We implement the provenance data collector of PROVDETECTOR which stores data in a PostgreSQL database using the Windows ETW framework [41] and Linux Audit framework [42]. The provenance graph builder and the representation extractor are implemented using about 15K lines of Java code, with the same method proposed by King et al. [12] and our causal path selection algorithm in §V-C. The rest parts of PROVDETECTOR, such as embedding and anomaly detection, are implemented in Python.

We use the $K = 20$ selected paths as the representation for a provenance graph. We then train a PV-DM model as discussed in §V-D using the Gensim library [43], which embeds each path into a 100 dimensional embedding vector, which is the default option of Gensim. Finally, we use the embedding vectors to train a novelty detection model using the Local Outlier Factor (LOF) algorithm in Scikit-learn [44].

Provenance Data Preprocessing Provenance data collected from different hosts may contain host-specific or entity-specific information such as file paths. To remove such information, we follow the abstraction rules that are similar to previous works [14], [15], [16]:

- *Path Abstraction*. Process entity and file entity have path related attributes such as process executable path and file path. We abstract these paths by removing user specific details. For example the path `C:/USERS/USER_NAME/DESKTOP/PAPER.DOC` will be changed to `*:/USERS/*/DESKTOP/PAPER.DOC`, where the user name and the root location are abstracted.
- *Socket Connection Abstraction*. A socket connection has two parts: the source part (IP and port) and the destination part (IP and port). As the IP of a host is a specific field only to the host, we abstract a socket connection by

removing the internal address while keeping the external address. More specifically, we remove the source part of an outgoing connection and the destination part of an incoming connection.

VI. Evaluation

To evaluate the efficacy of ProvDetector, we seek for answers to the following research questions:

RQ1: How effective is ProvDetector in detecting stealthy malware? What is the detection accuracy? (§VI-B)

RQ2: What makes ProvDetector capable of detecting stealthy malware? (§VI-C)

RQ3: What is the computational overhead of ProvDetector to build its models and to perform detection? (§VI-D)

A. Experiment Protocol

We answer the above three research questions with real stealthy malware instances gained by running malware samples and benign process instances gained from a real-world enterprise deployment. To collect benign provenance data, we installed the provenance data collector to 306 Windows hosts in an enterprise. The benign provenance data was collected over three months and stored in a PostgreSQL database.

To collect provenance data for stealthy malware, we downloaded about 15,000 malware samples from VirusShare [45] and VirusSign [46] and executed them in the Cuckoo sandbox [47]. Regarding the sandbox configuration, we prepared the same operating system (OS) and application environment as it is configured for the enterprise. Among the malicious execution instances, whose behaviors were triggered and captured by our sandbox, we identified 23 victim programs. These victims are benign programs used in the enterprise, whose behaviors are captured in the benign provenance dataset. The 23 hijacked victims include popular Windows applications such as IE Browser and Microsoft Word, and preinstalled system tools such as the Windows Certificate Services Tool. Table II shows the complete list.

In preparation of the dataset for model building, we chose 250 benign process instances and 50 malicious process instances for each of the 23 programs observed from both the benign and malicious environment. For each program, we randomly chose benign instances from the enterprise environment, whereas we generated corresponding malicious instances by running one distinct stealthy malware. In other words, we executed 50 distinct malware for each of the 23 programs to generate malicious data. Since our approach is an anomaly detection technique, which only needs benign data for training, we randomly selected 200 benign instances as the training dataset and used the rest 50 benign instances and all the malicious instances as the testing input. In total, we evaluated ProvDetector with 1,150 distinct malware samples that hijacks benign processes. These malware samples are classified into 189 malware families with AVClass [48]. Among the malware samples[2], 298 of them are identified to be anti-VM (*i.e.,* detecting if it is in a virtual machine) and 238 of them are identified to be anti-debug (*i.e.,* detecting if it is under debugger) by VirusTotal [49] or Tencent HABO [50].

We trained ProvDetector on a machine with an Intel Core i7-6700 Quad-Core Processor (3.4 GHz) and 32 GB RAM running Ubuntu 16.04 OS; detection was also performed on the same machine.

Removal of Biases Due to Sandbox Although we use real stealthy malware, the Cuckoo sandbox may introduce bias in our experiment. The workflow of how Cuckoo runs a stealthy malware is as follows: *(1)* the Cuckoo agent introduces a malicious payload (malware executable or malicious document) to the sandbox, *(2)* the initial payload injects malicious logic into a target benign program via various channels, *(3)* the injected malicious logic in the victim process executes. The first part of the workflow leaves a unique pattern in the provenance graphs due to the Cuckoo agent: every attack path in the provenance graph either starts with the agent process or the malicious payload. This pattern could introduce a bias to our experiment as the model can simply just remember the agent process or the malicious payload to predict whether a path is from a hijacked process. To eliminate such a bias, for all the malicious provenance graphs, we only use the sub-graph generated after the malicious payload has been loaded. In other words, we remove the event of loading the malicious payload and all other dependency events that happen before it. This pre-processing eliminates all the features related to the Cuckoo framework. To ensure that the generated provenance graphs do not have any bias, we examined the distribution of the embeddings of the paths generated from the benign workloads in the Cuckoo and confirmed that they follow the same distribution as our training data.

B. Detection Accuracy

To answer research question RQ1, we measure the detection accuracy for the 23 programs. To further evaluate the effectiveness of our proposed techniques, we also compare our embedding and anomaly detection methods to other baseline approaches.

In our experiments, we select the top 20 causal paths from each provenance graph using our path selection algorithm (§V-C1). Then, we measure both path-level detection accuracy and graph-level detection accuracy. To measure the path-level detection accuracy, we treat each path as an individual data sample; for the graph-level detection accuracy, we use the threshold-based method (§V-E) to make a final prediction from the predictions of paths. The detection accuracy of ProvDetector is measured using precision, recall, and F1-score metrics.

We show the path-level detection results in Table II. The detection accuracy of ProvDetector is consistently high across different programs. Precision ranges from 0.952 to 0.965, recall ranges from 0.965 to 1, and F1-score ranges from 0.961 to 0.982. We show the average graph-level detection accuracy for the 23 programs using different threshold values in Figure 5. Here the threshold value is the number of rarest paths selected as in §V-C. As we can see, using a threshold value of 3 or 4 already achieve very high precision and recall (precision of 0.957 and 0.995 for the threshold 3 and 4, respectively; recall of 1 for both of the threshold values 3

[2]We list the MD5 value of a malware, whether it is anti-VM, whether it is anti-debug and its AVClass label at https://github.com/share-we/malware.

TABLE II: The path-level detection accuracy of PROVDETECTOR.

Program	Description	Precision	Recall	F1-Score
attrib	Windows File System Tool	0.958	1	0.978
certutil	Windows Certificate Services Tool	0.964	1	0.981
cmd	Windows Command Line	0.956	0.999	0.977
cscript	Windows System Script Interpreter	0.959	0.999	0.978
cvtres	Component of C++ Toolchain	0.965	1	0.982
excel	Microsoft Excel	0.961	1	0.980
firefox	Firefox Browser	0.958	0.965	0.961
iexplore	IE Browser	0.960	0.968	0.963
javaw	Java VM	0.957	0.992	0.974
jusched	Java Update Scheduler	0.957	0.990	0.974
maintservice	Firefox Updater	0.959	1	0.979
msiexec	Windows Installer	0.960	0.983	0.971
mspaint	Microsoft Paint	0.96	0.990	0.975
notepad	Windows Text Editor	0.963	0.984	0.973
rar	WinRAR Compression Tool	0.953	1	0.976
sc	Windows Service Controller	0.952	1	0.975
spoolsv	Windows Spooler Subsystem App	0.955	1	0.977
tasklist	Windows Task Management Tool	0.962	0.970	0.966
taskmgr	Windows Task Manager	0.960	1	0.979
wget	Downloader	0.952	1	0.975
winword	Microsoft Word	0.960	0.976	0.967
wmic	Windows Management Instrumentation Command	0.952	0.998	0.974
wmplayer	Windows Media Player	0.959	0.996	0.977
Average	-	0.959	0.991	0.974

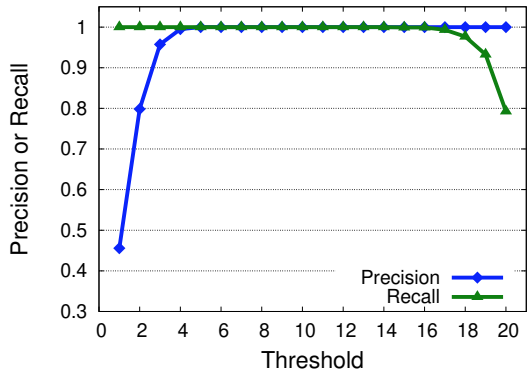


Fig. 5: The graph-level detection accuracy of PROVDETECTOR with different threshold values

and 4). All these results show that PROVDETECTOR is very effective in detecting stealthy malware.

1) Comparison with Strawman Detection Approaches: To show the effectiveness of our machine learning-based approach, we compare PROVDETECTOR with three strawman techniques: the blacklist, the whitelist, and the anomaly score based approach [14].

The goal of having the blacklist approach is to answer the question: is it possible to use hand-coded rules developed by human experts to detect stealthy attacks. Ideally, relying on human experts seems to be an effective approach which can easily bring in with several working heuristics. One exemplary

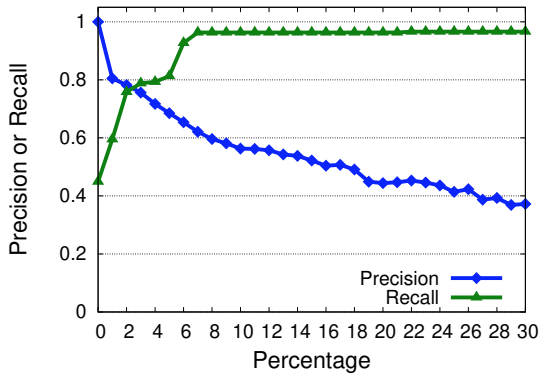


Fig. 6: The detection accuracy of the whitelist approach with different values.

rule can be "UI-heavy programs (*e.g.,* MS Word and Excel) should not launch external scripts, such as through CMD or PowerShell". However, in practice, since the adversary has a lot of ways to run the malicious code, it is very difficult to come up with a comprehensive blacklist. For instance, the UI-heavy processes could run the malicious code through Java or hijack other processes (*e.g.,* `notepad.exe`) instead of using scripts. Using a blacklist approach could overlook a large number of other attacks, especially unknown attacks. In our experiment, we measure the effectiveness of applying the "UI-heavy programs should not run external scripts" heuristic. To do so, we use all the 8 UI-heavy programs (*i.e.,* `excel`, `firefox`, `iexplore`, `mspaint`, `notepad`, `rar`, `winword` and `wmplayer`) in our evaluated 23 programs and check if they calls `cmd.exe`, `powershell.exe` or other script interpreters. We found that the recall of this heuristic is close to **zero** ($\leq$0.07), which means a large number of attacks were overlooked by this approach.

The second strawman approach, the whitelist, is to evaluate whether people can detect stealthy attacks by simply detecting infrequent events. To construct the whitelist, we use a statistics-based approach. For each event, if it exists in more than p percent of the benign program instances, we add it to the whitelist. In Figure 6, we show the detection accuracy of this approach averaged by the 23 programs using different p values. In our experiment, this approach achieves the best F1 score of 0.78 when p is 3%, which is still substantially lower than the F1 score of PROVDETECTOR.

The third strawman approach is the anomaly score-based approach. In §V-C, we define regularity score for a path to select the top K rarest paths from a provenance graph. One may consider that these regularity scores (or anomaly scores) could be used to effectively detect stealthy malware for simplicity. To address this concern, we evaluated a score-based detection approach. For each program, the anomaly score based approach first selects the top K rarest paths from all the benign provenance graphs, then it chooses the n-percentile of all the anomaly scores of the paths as the threshold. During the detection stage, it identifies any path that has an anomaly score higher than the threshold as a malicious path. In other words, if a path has an anomaly score higher than n percent of the paths selected from benign provenance graphs, this strawman approach identifies the path as malicious. The

TABLE III: The detection accuracy of the anomaly score based approach with different percentile values.

n-percentile	Precision	Recall	F1-Score
85	0.84	0.36	0.50
86	0.845	0.349	0.49
87	0.885	0.31	0.46
88	0.893	0.243	0.38
89	0.905	0.233	0.37
90	0.909	0.208	0.34
91	0.914	0.199	0.33
92	0.925	0.182	0.30
93	0.918	0.183	0.31
94	0.921	0.195	0.32
95	0.939	0.163	0.28

TABLE IV: Detection accuracy comparisons with path-level and graph-level approaches.

Path or Graph	Approach	Precision	Recall	F1-score
Path-level	ProvDetector (path-level)	**0.959**	0.991	0.974
	Path Nodes Averaging	0.961	0.890	0.924
Graph-level	ProvDetector (graph-level)	0.957	**1**	**0.978**
	graph2vec	0.899	0.452	0.601

results of detection accuracy with different percentile values are shown in Table III. The F1 score is even substantially lower than the whitelist approach. One major reason for such poor performance is that the rare paths selected from benign provenance graphs could also have very high anomaly scores. Therefore, the anomaly scores alone are not informative enough to differentiate benign ones and malicious ones. The results in Table III justify our choice of using a learning-based approach that learns both from rareness and causal dependencies to automatically identifies the proper boundary between benign and anomalous paths for each program.

2) Comparison with Different Embedding Approaches: We compare our embedding approach (§V-D) with a graph embedding approach (`graph2vec` [51]), and the simple node-level path embedding (Path Nodes Averaging). `graph2vec` is an approach to learn distributed representations of graphs. With `graph2vec`, we directly embed each provenance graph into a feature vector. In the Path Nodes Averaging approach, we still compute embeddings for the paths selected by ProvDetector. In contrast, we use `word2vec` to get the embedding of each node, then obtain the embedding for a path by averaging the embeddings of all the nodes in the path. In the evaluation of different embedding approaches, we follow the same experiment protocol in §VI-A.

To compare our approach with `graph2vec`, we compute graph-level detection accuracy of ProvDetector using a threshold of 3. The comparison results are shown in Table IV, in which ProvDetector has a substantially higher recall than `graph2vec`. The `graph2vec` approach has reasonable precision but has very poor recall (even worse than random guess). This result confirms our insight: the benign workloads of a hijacked process may hide the malicious workload in the graph level. It is thus necessary to use the path-level features. We will further discuss this result in §VI-C.

We compare our embedding approach with Path Nodes Averaging in path-level detection accuracy as shown in Table IV. The Path Nodes Averaging approach achieves comparable precision and recall with our approach as it also uses the paths selected by ProvDetector in the embedding. However, it does not perform as good as our approach on recall as it does not consider the order of nodes in a path.

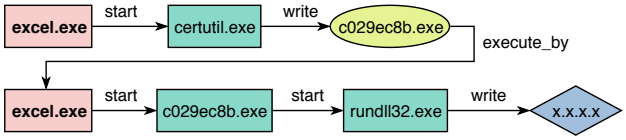


Fig. 7: The path selected by ProvDetector from a realistic attack example.

C. Interpretation of Detection Results

In this section, we interpret the detection results presented in §VI-B to justify our design decisions. In particular, we seek answers for the following questions:

- Why do simple models (*e.g.,* blacklist or whitelist) fail?
- Why the whole provenance graph is not a good feature for stealthy malware detection?
- Why our path selection method can accelerate the training and detection?
- How robust is ProvDetector against mimicry attacks?

For space reasons, we present other interpretations of the detection results in Appendix C.

1) Simple Models: To understand why simple models, such as the black- and white-lists, that only consider one-hop features are not effective, we use one realistic example in our experiment as a case study. The example is the "DownAuto Certutil Macro Dropper" malware, which is a part of APT28 attack [52], [53]. The causality chain of this attack is shown in Figure 7. This malware embeds its malicious payload as a base64 string and exploits the certificate services (`certutil.exe`) to convert the base64 string to an executable (`c029ec8b.exe`). After that, the malware runs the payload, which uses `rundll32.exe` to connect back to the adversary.

The analysis based on the one-hop relationships cannot disclose the adversarial context, as every step in this attack looks normal. It is possible for `excel.exe` to handle certificates with `certutil.exe` . It is also normal behavior for `certutil.exe` to create any arbitrary files. Note that in our experiment environment, although the malicious executable has a random name, in practice, this name could also be the name of any benign software and may not contain the extension `.exe`. Finally, it is also impractical to prevent `excel.exe` from executing external programs and `rundll32.exe` whose execution logic depends on its command line given at runtime. The abnormality of the operation arises only when all dots are connected and considered as a whole. ProvDetector models the whole causality path altogether as a vector and detects anomalous paths instead of anomalous steps. This is why ProvDetector outperforms simple approaches.

2) Whole Graph Modeling: To understand why the whole graph is not a good feature for detecting stealthy malware as well as why `graph2vec` does not perform well in §VI-B, we perform a set of empirical measurements. We randomly selected paths from the provenance graphs of processes that were hijacked by stealthy attacks. We then feed these paths into our anomaly detector to get their prediction. We found

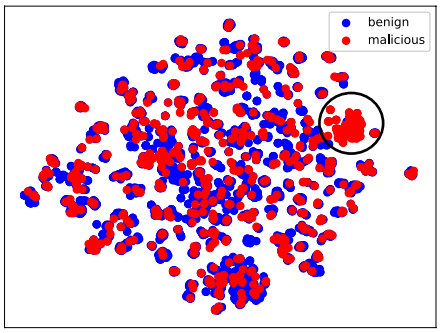


Fig. 8: The t-SNE plot with the paths randomly selected from benign and malicious provenance graphs of the `winword` program. The blue points and red points represent paths selected from benign provenance graphs and malicious provenance graphs respectively.

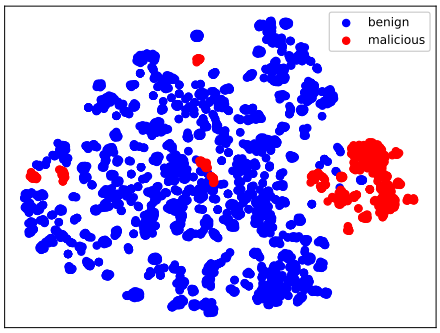


Fig. 9: The t-SNE plot with the paths selected by our path selection algorithm from benign and malicious provenance graphs of the `winword` program. The blue points and red points represent paths selected from benign provenance graphs and malicious provenance graphs respectively.

that, on average, about 70% of randomly selected paths from hijacked processes cannot be detected as malicious. In other words, about 70% of the paths are not distinguishable from benign paths. For a graph-level embedding method, which summarizes the features of all paths to get an embedding, will not be sensitive to a small number of abnormal paths.

To better understand the distribution of paths from hijacked programs, we take the `winword` (MS Word) program as an example and visualize the distribution in Figure 8. To generate Figure 8, we randomly select 20 paths from each provenance graph of `winword` (both benign and malicious), embed them with PROVDETECTOR, and plot the embedding vectors with t-SNE [54]. We mark the paths selected from benign graphs in blue and those from malicious graphs in red. In Figure 8, the majority of paths selected from malicious graphs are mixed with paths selected from benign graphs. This is because these "malicious" paths are generated from the benign part of the hijacked process. There is only a small group of paths that are easily separable, which we marked in a black circle. Therefore, graph-level embedding methods, such as `graph2vec`, which learn features from all the paths, is less capable of detecting stealthy malware as the features from "real" malicious paths are overlapped with the "normal" paths.

3) Path Selection: To demonstrate why our path selection technique can maintain the accuracy while reducing training and detection workload, we again take the `winword` program as an example. In Figure 9, we plot the embedding vectors of paths selected by PROVDETECTOR with t-SNE. The blue points are paths selected from benign provenance graphs and the red points are paths selected from malicious provenance graphs by PROVDETECTOR. The result in Figure 9 delivers two findings. First, the selected benign paths form multiple clusters representing the diversity of custom workloads of benign programs. Second, the selected (rare) paths from malicious graphs are very different from other benign paths, therefore they are easy to be separated in the embedding space. This result confirms our assumption that rare paths could capture abnormal behavior of stealthy malware.

4) Robustness Against Mimicry Attacks: The adversary may evade the detection of PROVDETECTOR by mimicking "normal" behaviors of programs. It is important to know how much effort does the adversary need to take to evade the detection.

To answer this question, we introduce the editing distance between malicious paths and benign paths. We define the editing distance between two causal paths as the minimum number of actions needed to convert one path to another. The actions include add, modify, and delete any node in a causal path[3]. In our experiment, we measured the average editing distance between malicious paths and benign paths[4]. The average value is about five. In other words, to make a malicious path looks benign, an adversary needs to mimic about five system objects. This result suggests that PROVDETECTOR is more robust than the single step detection approaches (*e.g.,* blacklist approach) since the adversary only needs to mimic the behavior of one system object.

D. Runtime Performance

We measure the runtime overhead of PROVDETECTOR for its training and detection stages.

Training Overhead The runtime overhead in the training stage for each monitored program mainly consists of *(1)* the overhead for building provenance graphs and path selection, *(2)* the overhead to build the `doc2vec` model, and *(3)* the overhead to build the anomaly detection model. On average, it takes seven seconds to build a provenance graph from the database and select the top 20 paths. With the data of 30,000 paths, it takes about 94 seconds to train the `doc2vec` model with the embedding vector size of 100 and epochs of 100. It takes around 39 seconds to train the LOF novelty detection model. Note that the training overhead for one program is a one-time effort. We do not need to retrain either of the

[3]This concept is borrowed from computational linguistics.

[4]To eliminate the bias introduced by arbitrary file names, we consider all files with the same type as one file; for network connection, we abstract all IPs to "*.*.*.*".

doc2vec model or the LOF model unless we want to improve the models with more training samples.

Detection Overhead The runtime overhead in the detection stage for a process instance mainly consists of *(1)* the overhead for building provenance graphs and path selection, *(2)* the overhead for embedding the selected paths, and *(3)* the prediction overhead of the anomaly detection model. On average, it takes five seconds to build the provenance graph and two seconds to select the top 20 paths from the graph. It only takes one millisecond (ms) to embed a path into a vector and 0.06 ms for the novelty detection model to make a prediction with the vector. In total, the detection overhead for a process instance is about seven seconds.

To estimate the practicality of PROVDETECTOR in an enterprise, we count the number of process instances created for the 23 evaluated programs from the data over three months with 306 hosts. On average, each host creates about 22.7 instances of these programs, *i.e.,* about one process for each program. Suppose an enterprise which has 100 hosts and there are 30 programs to monitor, it will take 5.7 hours per day to check all the created instances in the enterprise. However, note that our experiments were conducted on a single general desktop with a single thread. The detection time can be reduced by parallelizing PROVDETECTOR on multiple server machines.

VII. DISCUSSION AND LIMITATIONS

Offline Detection vs. Online Detection In our current implementation, PROVDETECTOR works as an offline detector, where it scans the provenance database to detect stealthy attacks. However, PROVDETECTOR can be implemented as a real-time approach by using an in-memory provenance graph database on each monitored host [55]. Then PROVDETECTOR can model the path selection problem as an incremental K longest paths problem on a dynamic graph, which is an orthogonal problem and has existing solutions [56], [57]. We leave the implementation details to our future work.

Applicability to Other Operating Systems In this work, our evaluation focuses on programs (*e.g.,* MS Word) on Windows systems as most of the stealthy malware we collected target Windows. However, our approach is not limited to a certain operating system like Windows since similar OS level provenance data can be also collected from other operating systems such as Linux [13]. Moreover, our approach does not rely on any Windows specific feature.

More Complex Embedding or Learning Approaches In this work, PROVDETECTOR uses the doc2vec paragraph embedding technique and a simple anomaly detection model LOF for its detection purpose. As shown in §VI, the combination of these two models have already achieved very good detection performance. More complex machine learning techniques, such as LSTM [58], Tree-structured LSTM [59], Graph Convolutional Networks [60], and One-class Neural Networks [61], [62] could possibly further improve the detection accuracy, yet they may also introduce a higher cost.

Mimicry Attacks An adversary may mimic behaviors of benign programs to evade the detection of PROVDETECTOR. In §VI-C4, we measured that, on average, an adversary needs to add, modify, or delete about five different nodes in a causal path to mimic the behavior of benign programs. Since a causal path embeds the contextual causality among different system entities (*e.g.,* processes), we believe that it is much harder to evade PROVDETECTOR than the approaches that focus only on the behavior of one process. We will conduct more evaluation and research on defending mimicry attacks in our future work.

Anti-analysis Malware A lot of today's malware has anti-analysis (*e.g.,* anti-VM or anti-debug) capabilities. When the malware detects that it is being run in a virtual machine or under a debugger, it changes its behavior (usually either less malicious behavior or termination). PROVDETECTOR, unlike virtualization based solutions [17], [18], is designed to run on bare metal machines and does not require isolated environments. Similar to previous work [63], [64], [18], to perform a large-scale analysis, we use sandbox environments to automate the execution of malware samples in our evaluation. It is possible that some anti-analysis malware changed their behavior during our evaluation. However, 289 (26%) of the malware samples in our evaluation are identified as anti-VM by VirusTotal. For these samples, PROVDETECTOR should still be able to detect them when they are running on bare metal machines as their behaviors on bare metal should be same or more malicious, which will be easily selected by PROVDETECTOR's path selection algorithm.

The Benign Dataset We collected our benign data from an anonymous enterprise which was well guarded by security professionals and continuously monitoring using up-to-date security solutions. Although it does not guarantee that our "benign" data is perfectly benign, we believe that the chance of data pollution is low and will not invalidate our evaluation.

VIII. RELATED WORK

Stealthy Malware Malware is becoming increasingly stealthy to evade detection. A popular trend in recent cyber-attacks is to impersonate or abuse benign applications on the victim host to achieve the attack goals. There are many impersonation techniques. For example, DLL injection [26], portable executable injection, and remote thread injection [65]. Recently developed new techniques such as process hollowing [27], AtomBombing [66] and shim-based DLL injection [28] have also been applied in real-world malware. Fileless malware, which follows the "living off the land" attack strategy, has been actively studied by both industry [9] and academia [67]. While characterized by its avoidance of using files during an attack, we believe that PROVDETECTOR will also be helpful in detecting certain types of fileless malware whose behavior can be tracked by our kernel-level provenance tracing.

Malware Detection Malware detection has been an active area of research in multiple platforms like Android, Windows, and Linux. In traditional approaches, static analysis [1], [2], [68] and dynamic analysis [69], [70], [71] have been used to analyze and detect malware. Recently, machine learning and deep learning approaches are leveraged as a new trend in malware analysis and detection which greatly improve the detection accuracy over traditional methods [3], [4], [5], [6]. Shu et al. [72] profile a program's historical behavior to detect stealthy control flow violations (*e.g.,* aberrant path

attack) based on function call logs gained by software instrumentation. Differently, PROVDETECTOR aims to detect a malware-controlled program using more coarse-grained kernel-provided audit logs. There are multiple proposals to detect stealthy malware that uses impersonation techniques like code injection. Bee master [63] prepares honeypot processes in an analysis environment and detects injections into the processes. Membrane [73] and Quincy [64] extract features from memory information such as memory paging information and memory dumps, and use supervised machine learning to detect code injection. Tartarus [17] and API Chaser [18] use taint tracking to identify code injection. However, these proposals either target only certain types of attacks [63] (*e.g.,* [63] cannot detect process hollowing), relay on some OS features [73], or need virtualization environments and have severe impact on the system performance [18], [17]. Moreover, all of them have a limitation for script-based attacks. In contrast, our approach uses lightweight kernel-level provenance tracking and targets the broad scope of impersonation techniques including script-based attacks.

Anomaly Detection with Host Level System Events Several approaches have been proposed to detect intrusion or abnormal behaviors using system event data on the end hosts [74], [75], [36], [22], [14]. Caselli el al. [74] proposed an approach which first builds the profile of k-grams from benign system call traces and then it throws an alert if a new system call trace is significantly different from the normal profile. Padmanabhan et al. [75] modeled the information flow in a system using directed graphs and extracts abnormal substructures from it. Dong et al. [36] proposed a system to find abnormal event sequences from a large number of heterogeneous event traces. Chen et al. [22] proposed a principled and unified probabilistic model to learn the likelihood of system events. Siddiqui et al. [76] developed a system to detect malicious system entities using a multi-view based technique.

Unstructured Data Embeddings Multiple embedding techniques (*i.e.,* learning distributed representations or numerical vectors of data) have been proposed for unstructured data such as texts and graphs. In the natural language processing domain, different embedding techniques have been proposed for words [77], [78], sentences [79] and documents [19]. Learning techniques have also been proposed for graphs [51], [80] as well. These embedding techniques are utilized in multiple security applications for data modeling. For example, Narayanan et al. [51] demonstrated the ability of `graph2vec` in classifying malicious and benign Android apps using API dependency graphs. Mimura et al. [81] used paragraph vectors to detect unseen malicious traffic from proxy log. Tavabi et al. [82] proposed a neural language modeling approach that learns embeddings of darkweb/deepweb discussions to predict whether vulnerabilities are exploited. In this work, we utilize paragraph embedding techniques over system provenance data to detect stealthy malware. PROVDETECTOR would benefit from the future improvement of embedding techniques.

Mimicry Attacks on Host-based Solutions System call traces have long been used as the information source for host-based instruction detection systems (IDS). The seminal research on mimicry attacks [83], [84] demonstrated that the IDS can be evaded by carefully crafting an exploit that produces a legitimate sequence of system calls while performing malicious actions. To limit the vulnerability of the IDS to mimicry attacks, a number of improvements [85], [86], [87], [88], [89] have been proposed by considering more features in the analysis. For example, [85] incorporates into the analysis information about the call stack configuration at the time of a system call to counteract mimicry attacks. To automate the construction of mimicry attacks, several techniques [90], [91], [92] have been proposed. However, these systems focus on monitoring system call traces, which do not reflect the context of each syscall event. In contrast, our approach uses data provenance that encodes historical context into causality graphs. Conducting mimicry attacks on provenance-based solutions is more challenging than on system call traces as provenance graphs contain complex structural information that is difficult to imitate without impeding the attack.

Provenance-based Solutions A large body of work has been proposed to leverage provenance for multiple areas such as forensic analysis [13], [38], [93], [33], [34], [16], [94], network debugging and troubleshooting [95], [96], alert triage [14], intrusion detection and access control [97], [98], [99], [15], [100], and attack reconstruction [101], [102], [103].

Linux Provenance Modules (LPM) [13] and Hi-Fi [34] proposed an efficient and trusted provenance collecting framework by adding provenance hooks in the Linux kernel similar to Linux Security Modules. BEEP [38] and ProTracer [33] are provenance trackers that solve the problem of dependency explosion in the provenance graph by execution partitioning the event-handling loops. Liu et al. [15] proposed an anomaly based priority search to address the dependency explosion problem. LogGC [93] further reduces the log size using the idea of execution partitioning. Winnower [16] provides a storage efficient provenance auditing framework for large clusters. MCI [104] proposes a reliable and efficient approach to restore fine-grained information flow among system events using dual execution (LDX). While these techniques address different problems, we believe that they can be integrated into PROVDETECTOR to improve its accuracy. Besides forensic investigation, provenance is also used in network debugging. Chen et al. [95] proposed differential provenance which reasons the differences compared to good and bad references. The same authors [96] also proposed secure packet provenance (SPP) that provides provenance on the Internet's data plane which has a high data rate. NoDoze [14] is an automated threat alert triage system based on data provenance. It ranks the alerts from third party threat detection systems (TDS) by the rareness of causal paths in their provenance graph. However, it cannot effectively extract the K rarest paths as it enumerates all the paths of a provenance graph. Moreover, it only provides anomaly scores to paths to help with investigation and does not provide a systematic way to separate benign and malicious paths. PROVDETECTOR addresses the limitations and provides an end-to-end solution to automatically learns the boundaries from training data using machine learning techniques. Besides, a TDS is not required by PROVDETECTOR.

IX. CONCLUSION

In this paper, we present PROVDETECTOR, an anomaly detection based approach to detect stealthy impersonation malware using OS level provenance graphs. PROVDETECTOR uses a novel rareness-based path selection algorithm to identify

causal paths in the provenance graph which represent the potentially malicious behavior of a process. These causal paths are then used by a pipeline of a document embedding model and a novelty detection model to determine if the process is malicious. We evaluated PROVDETECTOR with 23 target programs using a system provenance dataset from an enterprise. The results show that PROVDETECTOR has consistently high precision and recall for the evaluated programs, demonstrating its effectiveness and practicality in the detection of stealthy malware.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their helpful feedback. This work was supported in part by NSF CNS 13-30491. Ding Li and Kangkook Jee are the corresponding authors. The views expressed in this material are those of the authors only.

REFERENCES

[1] Monirul Sharif, Vinod Yegneswaran, Hassen Saidi, Phillip Porras, and Wenke Lee. Eureka: A framework for enabling static malware analysis. In *European Symposium on Research in Computer Security*, pages 481–500. Springer, 2008.

[2] A-D Schmidt, Rainer Bye, H-G Schmidt, Jan Clausen, Osman Kiraz, Kamer A Yuksel, Seyit Ahmet Camtepe, and Sahin Albayrak. Static analysis of executables for collaborative malware detection on android. In *2009 IEEE International Conference on Communications*, pages 1–5. IEEE, 2009.

[3] Zhenlong Yuan, Yongqiang Lu, Zhaoguo Wang, and Yibo Xue. Droid-sec: deep learning in android malware detection. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 371–372. ACM, 2014.

[4] Bojan Kolosnjaji, Apostolis Zarras, George Webster, and Claudia Eckert. Deep learning for classification of malware system call sequences. In *Australasian Joint Conference on Artificial Intelligence*, pages 137–149. Springer, 2016.

[5] William Hardy, Lingwei Chen, Shifu Hou, Yanfang Ye, and Xin Li. Dl4md: A deep learning framework for intelligent malware detection. In *Proceedings of the International Conference on Data Mining (DMIN)*, page 61. The Steering Committee of The World Congress in Computer Science, Computer . . . , 2016.

[6] Hamed HaddadPajouh, Ali Dehghantanha, Raouf Khayami, and Kim-Kwang Raymond Choo. A deep recurrent neural network based approach for internet of things malware threat hunting. *Future Generation Computer Systems*, 85:88–96, 2018.

[7] Cynthia Wagner, Alexandre Dulaunoy, Gérard Wagener, and Andras Iklody. Misp: The design and implementation of a collaborative threat intelligence sharing platform. In *Proceedings of the 2016 ACM on Workshop on Information Sharing and Collaborative Security*, pages 49–56. ACM, 2016.

[8] Trend Micro Research. 2019 Midyear Security Roundup: Evasive Threats, Pervasive Effects. Technical report, September 2019.

[9] The 2017 State of Endpoint Security Risk Report. https://www.barkly.com/ponemon-2018-endpoint-security-statistics-trends, 2018.

[10] Fileless Attack Survival Guide. https://dsimg.ubm-us.net/envelope/395823/551993/Fileless%20Attack%20Survival%20Guide.pdf, 2018.

[11] Matt Graeber. Abusing windows management instrumentation (wmi) to build a persistent, asyncronous, and fileless backdoor.

[12] Samuel T. King and Peter M. Chen. Backtracking intrusions. In *SOSP '03*. ACM, 2003.

[13] Adam Bates, Dave Tian, Kevin R. B. Butler, and Thomas Moyer. Trustworthy whole-system provenance for the linux kernel. In *Proceedings of the 24th USENIX Conference on Security Symposium*, pages 319–334, 2015.

[14] Wajih Ul Hassan, Shengjian Guo, Ding Li, Zhengzhang Chen, Kangkook Jee, Zhichun Li, and Adam Bates. NoDoze: Combatting threat alert fatigue with automated provenance triage. In *NDSS*, 2019.

[15] Yushan Liu, Mu Zhang, Ding Li, Kangkook Jee, Zhichun Li, Zhenyu Wu, Junghwan Rhee, and Prateek Mittal. Towards a timely causality analysis for enterprise security. In *NDSS*, 2018.

[16] Wajih Ul Hassan, Lemay Aguse, Nuraini Aguse, Adam Bates, and Thomas Moyer. Towards scalable cluster auditing through grammatical inference over provenance graphs. In *Network and Distributed Systems Security Symposium*, 2018.

[17] David Korczynski and Heng Yin. Capturing malware propagations with code injections and code-reuse attacks. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1691–1708, 2017.

[18] Yuhei Kawakoya, Eitaro Shioji, Makoto Iwamura, and Jun Miyoshi. Api chaser: Taint-assisted sandbox for evasive malware analysis. *Journal of Information Processing*, 27:297–314, 2019.

[19] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International conference on machine learning*, pages 1188–1196, 2014.

[20] Novelty detection with Local Outlier Factor. https://scikit-learn.org/stable/modules/outlier_detection.html#novelty-detection-with-local-outlier-factor, 2019.

[21] Yang Ji, Sangho Lee, Mattia Fazzini, Joey Allen, Evan Downing, Taesoo Kim, Alessandro Orso, and Wenke Lee. Enabling refinable cross-host attack investigation with efficient data flow tagging and tracking. In *USENIX Security Symposium*, 2018.

[22] Ting Chen, Lu-An Tang, Yizhou Sun, Zhengzhang Chen, and Kai Zhang. Entity embedding-based anomaly detection for heterogeneous categorical events. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, IJCAI'16, pages 1396–1403. AAAI Press, 2016.

[23] Peter F Brown, Peter V Desouza, Robert L Mercer, Vincent J Della Pietra, and Jenifer C Lai. Class-based n-gram models of natural language. *Computational linguistics*, 18(4):467–479, 1992.

[24] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE transactions on pattern analysis and machine intelligence*, 26(10):1367–1372, 2004.

[25] The 2017 State of Endpoint Security Risk. https://cdn2.hubspot.net/hubfs/468115/Campaigns/2017-Ponemon-Report/2017-ponemon-report-key-findings.pdf, 2017.

[26] Stephen Fewer. Reflective dll injection. *Harmony Security, Version*, 1, 2008.

[27] Process Hollowing. https://attack.mitre.org/techniques/T1093/, 2019.

[28] Defending Against Malicious Application Compatibility Shims. https://www.blackhat.com/docs/eu-15/materials/eu-15-Pierce-Defending-Against-Malicious-Application-Compatibility-Shims-wp.pdf, 2015.

[29] Microsoft Internet Explorer CVE-2019-0541 Remote Code Execution Vulnerability. https://www.symantec.com/security-center/vulnerabilities/writeup/106402, 2019.

[30] Macro-less Document and Fileless Malware: the perfect cloaking mechanism for new threats. https://forums.juniper.net/t5/Threat-Research/Macro-less-Document-and-Fileless-Malware-the-perfect-cloaking/ba-p/317425, 2018.

[31] PowerShell Empire. https://github.com/EmpireProject/Empire, 2019.

[32] Candid Wueest. Internet security threat report - living off the land and fileless attack techniques. 2017.

[33] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. ProTracer: Towards Practical Provenance Tracing by Alternating Between Logging and Tainting. In *NDSS*, 2016.

[34] D.J. Pohly, S. McLaughlin, P. McDaniel, and K. Butler. Hi-Fi: Collecting High-Fidelity Whole-System Provenance. In *ACSAC*, Orlando, FL, USA, 2012.

[35] Riccardo Paccagnella, Pubali Datta, Wajih Ul Hassan, Christopher W. Fletcher, Adam Bates, Andrew Miller, and Dave Tian. Custos: Practical tamper-evident auditing of operating systems using trusted execution. In *Proc. of the Symposium on Network and Distributed System Security (NDSS)*, 2020.

[36] Boxiang Dong, Zhengzhang Chen, Hui Wendy Wang, Lu-An Tang, Kai Zhang, Ying Lin, Zhichun Li, and Haifeng Chen. Efficient discovery of abnormal event sequences in enterprise security systems.

In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, pages 707–715. ACM, 2017.

[37] Salima Omar, Asri Ngadi, and Hamid H Jebur. Machine learning techniques for anomaly detection: an overview. *International Journal of Computer Applications*, 79(2), 2013.

[38] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. High Accuracy Attack Provenance via Binary-based Execution Partition. In *NDSS*, 2013.

[39] A Bako. All paths in an activity network. *Statistics: A Journal of Theoretical and Applied Statistics*, 7(6):851–858, 1976.

[40] David Eppstein. Finding the k shortest paths. *SIAM Journal on computing*, 28(2):652–673, 1998.

[41] Event Tracing. https://docs.microsoft.com/en-us/windows/desktop/ETW/event-tracing-portal, 2019.

[42] System administration utilities, 2019. man7.org/linux/man-pages/man8/auditd.8.html/.

[43] gensim: Topic modelling for humans. https://radimrehurek.com/gensim/index.html, 2019.

[44] scikit-learn: machine learning in Python. https://scikit-learn.org/, 2019.

[45] VirusShare. https://virusshare.com, 2019.

[46] VirusSign. https://www.virussign.com/, 2019.

[47] Cuckoo Sandbox - Automated Malware Analysis. https://cuckoosandbox.org/, 2019.

[48] Marcos Sebastián, Richard Rivera, Platon Kotzias, and Juan Caballero. Avclass: A tool for massive malware labeling. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 230–253. Springer, 2016.

[49] VirusTotal. https://www.virustotal.com/, 2018.

[50] Tencent HABO. https://habo.qq.com/, 2019.

[51] Annamalai Narayanan, Mahinthan Chandramohan, Rajasekar Venkatesan, Lihui Chen, Yang Liu, and Shantanu Jaiswal. graph2vec: Learning distributed representations of graphs. *arXiv preprint arXiv:1707.05005*, 2017.

[52] VirusTotal Report. https://www.virustotal.com/gui/file/56f98e3ed00e48ff9cb89dea5f6e11c1/, 2019.

[53] Sofacy Attacks Multiple Government Entities. https://unit42.paloaltonetworks.com/unit42-sofacy-attacks-multiple-government-entities/, 2019.

[54] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.

[55] Md Nahid Hossain, Sadegh M. Milajerdi, Junao Wang, Birhanu Eshete, Rigel Gjomemo, R. Sekar, Scott D. Stoller, and V. N. Venkatakrishnan. SLEUTH: real-time attack scenario reconstruction from COTS audit data. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.*, pages 487–504, 2017.

[56] Irit Katriel, Laurent Michel, and Pascal Hentenryck. Maintaining longest paths incrementally. *Constraints*, 10(2):159–183, April 2005.

[57] Tanya Y Berger-Wolf and Jared Saia. A framework for analysis of dynamic social networks. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 523–528. ACM, 2006.

[58] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[59] Kai Sheng Tai, Richard Socher, and Christopher D Manning. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*, 2015.

[60] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[61] Raghavendra Chalapathy, Aditya Krishna Menon, and Sanjay Chawla. Anomaly detection using one-class neural networks. *arXiv preprint arXiv:1802.06360*, 2018.

[62] Pramuditha Perera and Vishal M Patel. Learning deep features for one-class classification. *IEEE Transactions on Image Processing*, 2019.

[63] Thomas Barabosch, Sebastian Eschweiler, and Elmar Gerhards-Padilla. Bee master: Detecting host-based code injection attacks. In *International conference on detection of intrusions and malware, and vulnerability assessment*, pages 235–254. Springer, 2014.

[64] Thomas Barabosch, Niklas Bergmann, Adrian Dombeck, and Elmar Padilla. Quincy: Detecting host-based code injection attacks in memory dumps. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 209–229. Springer, 2017.

[65] Remote Thread Injection on Windows. http://blog.aaronballman.com/2011/06/remote-thread-injection-on-windows/, 2011.

[66] AtomBombing: Brand New Code Injection for Windows. https://blog.ensilo.com/atombombing-brand-new-code-injection-for-windows, 2016.

[67] Fan Dang, Zhenhua Li, Yunhao Liu, Ennan Zhai, Qi Alfred Chen, Tianyin Xu, Yan Chen, and Jingyu Yang. Understanding fileless attacks on linux-based iot devices with honeycloud. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, pages 482–493. ACM, 2019.

[68] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 576–587. ACM, 2014.

[69] Blake Anderson, Daniel Quist, Joshua Neil, Curtis Storlie, and Terran Lane. Graph-based malware detection using dynamic analysis. *Journal in computer Virology*, 7(4):247–258, 2011.

[70] Anusha Damodaran, Fabio Di Troia, Corrado Aaron Visaggio, Thomas H Austin, and Mark Stamp. A comparison of static, dynamic, and hybrid analysis for malware detection. *Journal of Computer Virology and Hacking Techniques*, 13(1):1–12, 2017.

[71] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 15–26. ACM, 2011.

[72] Xiaokui Shu, Danfeng Yao, and Naren Ramakrishnan. Unearthing stealthy program attacks buried in extremely long execution paths. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 401–413, 2015.

[73] Gábor Pék, Zsombor Lázár, Zoltán Várnagy, Márk Félegyházi, and Levente Buttyán. Membrane: a posteriori detection of malicious code loading by memory paging analysis. In *European Symposium on Research in Computer Security*, pages 199–216. Springer, 2016.

[74] Marco Caselli, Emmanuele Zambon, and Frank Kargl. Sequence-aware intrusion detection in industrial control systems. In *Proceedings of the 1st ACM Workshop on Cyber-Physical System Security*, pages 13–24. ACM, 2015.

[75] Kanchana Padmanabhan, Zhengzhang Chen, Sriram Lakshminarasimhan, Siddarth Shankar Ramaswamy, and Bryan Thomas Richardson. Graph-based anomaly detection. *Practical Graph Mining with R (2013)*, 311, 2013.

[76] Md Amran Siddiqui, Alan Fern, Ryan Wright, Alec Theriault, David Archer, and William Maxwell. Detecting cyberattack entities from audit data via multi-view anomaly detection with feedback. In *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[77] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[78] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146, 2017.

[79] Matteo Pagliardini, Prakhar Gupta, and Martin Jaggi. Unsupervised learning of sentence embeddings using compositional n-gram features. *arXiv preprint arXiv:1703.02507*, 2017.

[80] Dang Nguyen, Wei Luo, Tu Dinh Nguyen, Svetha Venkatesh, and Dinh Phung. Learning graph representation via frequent subgraphs. In *Proceedings of the 2018 SIAM International Conference on Data Mining*, pages 306–314. SIAM, 2018.

[81] Mamoru Mimura and Hidema Tanaka. A linguistic approach towards intrusion detection in actual proxy logs. In *International Conference on Information and Communications Security*, pages 708–718. Springer, 2018.

[82] Nazgol Tavabi, Palash Goyal, Mohammed Almukaynizi, Paulo Shakarian, and Kristina Lerman. Darkembed: Exploit prediction with neural language models. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[83] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*. ACM, 2002.

[84] Debin Gao, Michael K Reiter, and Dawn Song. On gray-box program tracking for anomaly detection. *Department of Electrical and Computing Engineering*, page 24, 2004.

[85] Henry Hanping Feng, Oleg M Kolesnikov, Prahlad Fogla, Wenke Lee, and Weibo Gong. Anomaly detection using call stack information. In *2003 Symposium on Security and Privacy*. IEEE, 2003.

[86] Jonathon T Giffin, Somesh Jha, and Barton P Miller. Efficient context-sensitive intrusion detection. In *NDSS*, 2004.

[87] Jonathon T Giffin, David Dagon, Somesh Jha, Wenke Lee, and Barton P Miller. Environment-sensitive intrusion detection. In *International Workshop on Recent Advances in Intrusion Detection*, pages 185–206. Springer, 2005.

[88] Federico Maggi, Matteo Matteucci, and Stefano Zanero. Detecting intrusions through system call sequence and argument analysis. *IEEE Transactions on Dependable and Secure Computing*, 7(4):381–395, 2008.

[89] Kui Xu, Ke Tian, Danfeng Yao, and Barbara G Ryder. A sharper sense of self: Probabilistic reasoning of program behaviors for anomaly detection with context sensitivity. In *DSN*. IEEE, 2016.

[90] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. Automating mimicry attacks using static binary analysis. In *USENIX Security Symposium*, volume 14, 2005.

[91] Jonathon T Giffin, Somesh Jha, and Barton P Miller. Automated discovery of mimicry attacks. In *International Workshop on Recent Advances in Intrusion Detection*, pages 41–60. Springer, 2006.

[92] Chetan Parampalli, R Sekar, and Rob Johnson. A practical mimicry attack against powerful system-call monitors. In *Proceedings of the 2008 ACM symposium on Information, computer and communications security*. ACM, 2008.

[93] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. LogGC: Garbage collecting audit log. In *CCS*, pages 1005–1016, New York, NY, USA, 2013. ACM.

[94] Adam Bates, Wajih Ul Hassan, Kevin Butler, Alin Dobra, Bradley Reaves, Patrick Cable, Thomas Moyer, and Nabil Schear. Transparent web service auditing via network provenance functions. In *WWW*, 2017.

[95] Ang Chen, Yang Wu, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. The Good, the Bad, and the Differences: Better Network Diagnostics with Differential Provenance. In *ACM SIGCOMM*, 2016.

[96] Ang Chen, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. One primitive to diagnose them all: Architectural support for internet diagnostics. In *EuroSys*, 2017.

[97] Adam Bates, Kevin R. B. Butler, and Thomas Moyer. Take Only What You Need: Leveraging Mandatory Access Control Policy to Reduce Provenance Storage Costs. In *TaPP*, 2015.

[98] Jaehong Park, Dang Nguyen, and Ravi Sandhu. A provenance-based access control model. In *Privacy, Security and Trust (PST), 2012 Tenth Annual International Conference on*, pages 137–144. IEEE, 2012.

[99] Dang Nguyen, Jaehong Park, and Ravi Sandhu. Adopting provenance-based access control in openstack cloud iaas. In *International Conference on Network and System Security*, pages 15–27. Springer, 2014.

[100] Benjamin E Ujcich, Samuel Jero, Anne Edmundson, Qi Wang, Richard Skowyra, James Landry, Adam Bates, William H Sanders, Cristina Nita-Rotaru, and Hamed Okhravi. Cross-app poisoning in software-defined networking. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 648–663, 2018.

[101] Xingzi Yuan, Omid Setayeshfar, Hongfei Yan, Pranav Panage, Xuetao Wei, and Kyu Hyung Lee. DroidForensics: Accurate reconstruction of android attacks via multi-layer forensic logging. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, 2017.

[102] Qi Wang, Wajih Ul Hassan, Adam Bates, and Carl Gunter. Provenance tracing in the internet of things. In *Proceedings of the 9th USENIX Conference on Theory and Practice of Provenance*, pages 9–9, 2017.

[103] Qi Wang, Wajih Ul Hassan, Adam Bates, and Carl Gunter. Fear and logging in the internet of things. In *Network and Distributed Systems Symposium*, 2018.

[104] Yonghwi Kwon, Fei Wang, Weihang Wang, Kyu Hyung Lee, Wen-Chuan Lee, Shiqing Ma, Xiangyu Zhang, Dongyan Xu, Somesh Jha, Gabriela F Ciocarlie, et al. Mci: Modeling-based causality inference in audit logging for attack investigation. In *NDSS*, 2018.

[105] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. Isolation forest. In *2008 Eighth IEEE International Conference on Data Mining*, pages 413–422. IEEE, 2008.

[106] Bernhard Schölkopf, Robert C Williamson, Alex J Smola, John Shawe-Taylor, and John C Platt. Support vector method for novelty detection. In *Advances in neural information processing systems*, pages 582–588, 2000.

[107] Fitting an elliptic envelope. https://scikit-learn.org/stable/modules/outlier_detection.html#fitting-an-elliptic-envelope, 2019.

[108] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "why should I trust you?": Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, pages 1135–1144, 2016.

APPENDIX

A. Neural Document Embedding Models

Word2vec [77] is one of the most well-known word embedding methods. It uses a simple and efficient feed forward neural network architecture called "skip-gram" to learn distributed representations of words. Recently, Le and Mikolov proposed Paragraph Vector (*i.e.,* `doc2vec`) [19], a straightforward extension of `word2vec` that is capable of learning distributed representations of arbitrary length word sequences such as sentences, paragraphs and even whole large documents.

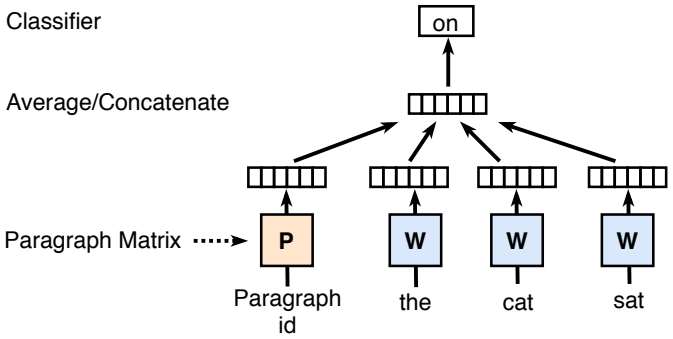


Fig. 10: The PV-DM model for learning a paragraph vector.

PV-DM (Distributed Memory Model of Paragraph Vectors) is one version of `doc2vec`. The core idea of PV-DM is that a paragraph p can be represented as another vector (*i.e.,* paragraph vector) contributing to the prediction of the next word in a sentence. In the PV-DM model as illustrated in Figure 10, every paragraph is mapped to a paragraph vector, represented by a column in a paragraph matrix and every word is mapped to a word vector, represented by a column in a word matrix. Then the paragraph vector and word vectors are averaged or concatenated to predict the next word in a context. The contexts are fixed-length and sampled from a sliding

TABLE V: Comparison of different anomaly detection algorithms in path-level detection accuracy.

Algorithm	Precision	Recall	F1-Score
Local Outlier Factor	**0.959**	**0.991**	**0.974**
One-Class SVM	0.886	0.635	0.739
Isolation Forest	0.955	0.467	0.627
Robust Covariance	0.940	0.397	0.558

window over the paragraph. The paragraph vector is shared across all contexts generated from the same paragraph but not across paragraphs. The PV-DM model uses stochastic gradient descent to train the paragraph vectors and word vectors. After being trained, the paragraph vectors can be used as features for the paragraph. At prediction time, the model also use gradient descent to compute the paragraph vector for a new paragraph.

B. Comparison of Different Anomaly Detection Algorithms

In our current implementation, we use Local Outlier Factor (LOF) [20] as the default anomaly detector. We compare LOF with three other novelty detection or outlier detection algorithms in path-level accuracy. The three baseline methods are as follows:

- Isolation Forest [105]: This algorithm divides the data points to different partitions. Outliers need less cuts to be separated from other points while inliers need more cuts.
- One-Class SVM [106]: The algorithm trains a hyper-plane which separates all the training data from the origin while maximizing the distance from the origin to the hyper-plane.
- Robust Covariance (Elliptic Envelope) [107]: The algorithm assumes that the data is Gaussian distribution and learns an ellipse.

In the evaluation of the above baseline methods, we follow the same experiment protocol as we did for PROVDETECTOR . For one-class SVM, we use the `rbf` kernel with `nu` set to 0.1 and `gamma` set to 0.5. For the other three models, we set the contamination to 0.04. The results are summarized in Table V.

As shown in the table, LOF significantly outperforms other methods in terms of recall. This justifies our design choice of using LOF. We will further explain the results in §VI-C.

C. Additional Experiments to Interpret the Result of PROVDETECTOR

In this section, we discuss several additional questions about the results of PROVDETECTOR. These questions are:

- Why LOF outperforms the other three anomaly detection methods?
- What is learned by PROVDETECTOR?

1) Why does LOF Perform Better: As shown in Table V, LOF performs the best among the four evaluated algorithms. This is because LOF does not rely on an assumption about the distribution of the data. As shown in Figure 9, the embeddings of paths have multiple clusters and do not follow any single distribution.

Robust Covariance performs worst as it assumes the data obeys approximately a Gaussian distribution and tries to learn an ellipse to cover the normal data points. Consequently, it may degrade when the data is not unimodal. Isolation Forest and One-Class SVM outperform Robust Covariance because they do not rely on any assumption on the distribution of data. However, these two methods assume that the normal paths are all from one cluster; thus they cannot achieve high detection accuracy as high as LOF.

On the other hand, LOF detects anomalous data points by measuring the local deviation of a given data point with respect to its neighbors, making it typically suitable for the case where different models in the data have different densities. As with our data, different workloads may generate paths that have different densities in distribution, thus LOF could achieve a high detection accuracy.

2) What PROVDETECTOR *Learns:* There are two possible kinds of features that PROVDETECTOR has learned: the path-level feature or the single node level feature. If PROVDETECTOR only learns single node level features, it could indicate that PROVDETECTOR only memorizes a small set of nodes to detect malicious paths. Still take the `winword` program as an example, a "bad" detection model which only learns node level features might predict a path as malicious if a previously unseen process (*e.g.,* PowerShell) node is in the path. Such detection model can easily be evaded by attackers.

To answer this question, a naive method is to develop baseline detection methods that only rely on single node level features. However, this method may have a bias from what baseline detection methods we select. Instead, we use LIME [108], a model-agnostic prediction explanation tool, to calculate and rank the "impact" of each single node in a path to the final detection. LIME also produces a numeric value to evaluate how much the final result would change, in case we remove any node from the path.

We use LIME to calculate the "*KEY*" nodes for each benign path and malicious path. A set of nodes are considered as KEY nodes if they are the most impactful nodes identified by LIME and PROVDETECTOR would give a different detection result if we remove these nodes from the path. We try to find if there is a set of KEY nodes that are common across all the paths. If so, it indicates that PROVDETECTOR has only learned single node level features.

In our experiment, we find that there is *not* a set of KEY nodes that can be shared by most of the paths. For benign paths, 35% of the paths have their own unique KEY node. On average, the number of paths that share the same KEY node is 3.18. In other words, each KEY node is used to impact 3 benign paths on average in PROVDETECTOR. For malicious paths, about 50% of paths have their unique KEY node. The average number of paths that share the same KEY node is 3.1. In summary, combined with the results in §VI-B2, PROVDETECTOR relies on path-level features instead of single node level features to detect stealthy malware, which is consistent with our design motivation.