Graphite: Real-Time Graph-Based Detection of Windows Fileless Malware Attacks

Priti Wakodikar[1(✉)], Joon-Young Gwak[1(✉)], Meng Wang[1], Guanhua Yan[1], Xiaokui Shu[2], Scott Stoller[3], and Ping Yang[1]

[1] School of Computing, Binghamton University, Binghamton, USA
{pwakodi1,jgwak1,mwang105,ghyan,pyang}@binghamton.edu
[2] IBM Thomas J. Watson Research Center, Yorktown Heights, USA
xiaokui.shu@ibm.com
[3] Department of Computer Science, Stony Brook University, Stony Brook, USA
stoller@cs.stonybrook.edu

Abstract. Advanced malware attacks often employ sophisticated tactics such as DLL injection, script-based attacks, and the exploitation of zero-day vulnerabilities. As evidenced by the recent high-profile cyber-attacks, these techniques have enabled attackers to infiltrate computer systems that were thought to be well-protected. There is thus an urgent need to enhance current malware defenses with advanced Artificial Intelligence (AI) techniques that can effectively detect in real-time the elusive traces of malware attacks concealed within the extensive realm of normal activities. This paper introduces *Graphite*, a graph-based approach for real-time detection of advanced malware attacks based on the event data collected from Event Tracing for Windows (ETW). Graphite first abstracts various entities and their relationships embodied within system events into computation graphs, which are amenable to graph-based machine learning methods. As a computation graph can be gigantic, making real-time malware detection inefficient, we project the graph into smaller graphlets, which are then subsequently fed into our graph-based approach to detect malicious activities. Our experimental results show that Graphite achieves 87.7% classification accuracy in offline testing and 86.7% accuracy in real-time detection.

Keywords: Malware detection · Machine learning

1 Introduction

The current Internet has been inundated by numerous malware attacks. In 2022, there were 5.4 billion malware attacks worldwide [65] with over 270,000 identified as new malware variants in the first half of the year alone, an increase of 45%

P. Wakodikar and J.-Y. Gwak—Co-first authors.

S. Alrabaee et al. (Eds.): SecureComm 2024, LNICST 629, pp. 154–178, 2026.
https://doi.org/10.1007/978-3-031-94455-0_8

over the same period in the previous year [66]. This influx of malware attacks has been contributing to a wide spectrum of online criminal activities, including cyber extortion, intellectual property theft, identity theft, email spamming, etc.

In response to advances in malware defense mechanisms, fileless malware have emerged as an effective attack vector to compromise targeted computer systems in recent years. Unlike traditional malware that uses custom executable programs, fileless malware attacks leverage legitimate system utilities such as PowerShell and Windows Management Instrumentation (WMI) to perform malicious activities [12,40]. Such attacks do not require to install new malware code, and hence leave little trace behind, which allows them to bypass even the most up-to-date malware detection systems. A recent report [26] shows a $1,400\%$ surge in fileless malware attacks from the previous year, underscoring their emergence as a popular threat vector in today's cybersecurity landscape.

The literature outlines three main approaches for detecting fileless malware attacks. Fileless malware attacks sometimes utilize in-memory execution through built-in tools such as PowerShell [12], evading detection by traditional antivirus programs that scan for malicious files. Consequently, *Memory-based detection* was introduced to scan computer memory for traces of such attacks [36,64]. However, these techniques often incur high execution overhead due to continuous memory scanning [8] and their detection accuracy can suffer against evasive malware attacks due to reliance on known malware signatures [60]. Various *static script analysis* methods were proposed to identify malicious intentions in the scripts developed for these platforms [27,28,46]. However, due to their static analysis nature, these methods can sometimes be thwarted by sophisticated obfuscation techniques [15,43]. A vast body of *provenance-based detection* techniques have been proposed to construct provenance graphs from events logged by computer systems, which are analyzed to identify sophisticated cyberattack operations, including fileless malware attacks [31,32,34,44,61,62]. While many of these efforts focus on postmortem attack scenario reconstruction, some apply detection rules on the provenance graphs to identify malicious attacks in real time. To be effective, these rules need to be manually crafted based on in-depth knowledge about sophisticated cyberattack operations and be regularly updated to adapt to evolving attack strategies.

Against this backdrop, this work aims to develop new techniques for the automatic detection of fileless malware attacks. Our work builds upon provenance-based detection methods, known for their greater resilience against evasion attacks than memory-based detection methods. To overcome the challenge of crafting effective policy rules for malware detection based on provenance graphs, we proposed a graph-based machine learning method to automatically detect malicious activities at runtime.

As the majority of malware attacks (including fileless ones) target computer systems running Microsoft Windows, we propose a new framework called *Graphite* to detect such attacks from system events collected by Event Tracing for Windows (ETW), the high-speed tracing and logging facility built into Windows systems [74]. Utilizing ETW logs, Graphite abstracts various entities, such

as processes and files, and their relationships embodied within system events into *computation graphs* [62], which are amenable to graph-based machine learning methods. Unlike provenance graphs that are process-centered, our computation graphs are thread-centered graphs, which provides finer-grained information specific to each thread. Our experimental results in Sect. 5 demonstrate that leveraging local temporal information associated with each thread, rather than the global temporal information associated with the entire process, results in higher classification accuracy.

Several technical challenges hinder the direct application of graph-based learning models to computation graphs. Firstly, the graph representation of extensive low-level system event logs can be gigantic, rendering the direct application of graph-based learning models inefficient. For instance, a graph constructed from more than 100 million system events collected from six computers over a two-week period comprises over 13 million nodes, requiring 161 GB storage space [62]. Secondly, due to the diversity of system events within a computer system, nodes and edges in their graph representations are commonly augmented with heterogeneous attributes. For example, a node may represent a process, a thread, a file, and more. Overlooking these informative attributes during the training of a graph-based learning model may lead to significant degradation in predictive accuracy. Finally, due to the inherent dynamic nature of system events, their graph representations undergo constant changes over time. Consequently, the temporal evolution introduces an additional layer of intricacy to the challenge of training predictive graph models for real-time malware detection.

To address the above challenges, we develop a projection algorithm to decompose a large computation graph into small graphlets. Using a technique inspired by taint analysis, this algorithm captures all the activities of a process and its descendant processes or threads. The graphlets are fed to a downstream classification model for fileless malware detection. Graphite uses a graph-based learning model that leverages temporal and spatial information in the graphlets to update node embeddings to perform malware classification.

Due to the challenges in acquiring comprehensive datasets for all types of fileless attacks, this paper focuses on those leveraging PowerShell scripts. Our approach is also applicable to other forms of fileless attacks. The source code of Graphite and the dataset used in this paper is available at[1].

In summary, our contributions are outlined as follows:

- We collected event logs of benign and malicious PowerShell scripts downloaded from various sources using ETW and proposed a schema to characterize the relationships between different subject and object entities in ETW logs. Based on this schema, we developed algorithms to construct computation graphs with various node-level and edge-level attributes. Unlike provenance graphs that are process centered, our computation graphs are thread-centered, providing finer-grained information specific to each thread.
- Inspired by taint analysis, we develop a projection algorithm that decomposes a large computation graph into small graphlets, each of which encompasses

[1] https://github.com/jgwak1/Graphite.

relevant system events initiated by a process and all its descendant processes and threads.
- We developed a graph-based machine learning model to distinguish between malicious and benign graphlets. Leveraging the central role of thread nodes in the schema, our method utilizes temporal and spatial information within graphlets to update graph embeddings for malware classification. Our experimental results show that Graphite achieves a classification accuracy of 87.7% on the dataset we collected.
- We developed a real-time malware detection algorithm capable of detecting malware based on partial graphlets whose size is determined by a user-defined threshold. Our experimental results show a real-time detection accuracy of 86.7%, slightly lower than offline testing results.

Organization: The rest of the paper is organized as follows. Section 2 provides an overview of ETW and fileless PowerShell malware. Section 3 presents the design and architecture of Graphite. Section 4 introduces our graph-based malware detection technique. The experimental results are detailed in Sect. 5. Section 6 summarizes the related work and Sect. 7 concludes the paper.

2 Background

This section provides an overview of Event Tracing for Windows (ETW) and fileless PowerShell malware.

2.1 Event Tracing for Windows (ETW)

ETW offers facilities to trace and log events raised by both user-mode applications and kernel-mode drivers within the Windows Operating System [74]. ETW records system calls related to I/O operations, including file reads and writes, as well as process operations such as process start and termination. ETW has many different event providers for tracing different types of events.

We utilize a C# wrapper for ETW called SilkService [23] to collect event logs. Figure 1 gives an example of a log entry generated by the "Microsoft-Windows-Kernel-Process" provider. Each entry in an ETW log provides basic information such as the name and ID of the event provider, and the time when the event occurred (i.e., TimeStamp). Other keys in the event log are specific to the type of provider, such as file, process, registry, or network. For instance, the event "ThreadStop/Stop" in the figure specifies that a process thread with "ThreadID" 11996 has stopped. The corresponding process for this thread has "ProcessID" 8416.

2.2 Fileless PowerShell Malware

Living-off-the-land techniques that use built-in system tools such as PowerShell, WMI, Command Prompt, and batch scripts, are common in APTs and targeted

{ "ProviderGuid": "22fb2cd6-0e7b-422b-a0c7-2fad1fd0e716", "YaraMatch": [ ], "ProviderName": "Microsoft-Windows-Kernel-Process", "EventName": "ThreadStop/Stop", "Opcode": 2, "OpcodeName": "Stop", "TimeStamp": "2019-03-03T17:58:14.2862348+00:00", "ThreadID":11996, "ProcessID":8416, "ProcessName": "", "PointerSize":8, "EventDataLength":76, "XmlEventData":{ "FormattedMessage": "Thread 11,996 (in Process 8,416) stopped.", "StartAddr": "0x7fffe299a110", "ThreadID": "11,996", "UserStackLimit": "0x3d632000", "StackLimit": "0xfffff38632d39000", "MSec": "560.5709", "TebBase": "0x91c000", "CycleTime": "4,266,270", "ProcessID": "8,416", "PID": "8416", "StackBase": "0xfffff38632d40000", "SubProcessTag": "0", "TID": "11996", "ProviderName": "Microsoft-Windows-Kernel-Process", "PName": "", "UserStackBase": "0x3d640000", "EventName": "ThreadStop/Stop", "Win32StartAddr": "0x7fffe299a110" } }

Fig. 1. Example of an ETW event log entry.

attacks. These tools facilitate lateral movement, data exfiltration, and privilege escalation. Our research primarily focuses on utilizing PowerShell, a scripting language for automating tasks on Windows machines. PowerShell provides a command prompt interface and facilitates task automation through simple scripts. These scripts contain commands referred to as "cmdlets", each performing a distinct task. The output of one cmdlet can be used as the input to subsequent commands.

PowerShell scripts has been utilized by fileless malware to carry out malicious activities with little trace left behind. A typical scenario is the fileless PowerShell script attack, in which the attacker leverages PowerShell's capabilities to execute commands directly in memory to gain full access to the Windows API, enabling system exploitation while bypassing the need to save the script on disk. This makes it difficult to detect fileless PowerShell scripts using traditional file-based antivirus solutions. Some of the PowerShell scripts used in our experiments, such as Invoke-DllInjection.ps1 [75], Invoke-ReflectivePEInjection.ps1 [52], Invoke-PSInject.ps1 [52] also inject malicious code into a hijacked process.

Fileless malware attackers employ a variety of techniques to execute suspicious activities. To evade detection, they frequently resort to obfuscation methods such as encryption or encoding within PowerShell code. This practice presents a challenge for static code analysis aimed at deciphering the true intent behind such actions. A PowerShell script "Out-EncodedAsciiCommand.ps1" serves as an example of these obfuscation tactics [6]. The Out-EncodedAsciiCommand PowerShell function encodes a provided PowerShell script block or path into an ASCII payload. Its execution involves multiple parameters, including "Script block", where payload specifications are defined. Executing these obfuscated PowerShell scripts often entails utilizing built-in Microsoft commands such as "Get-Information.ps1" [47], which are employed to gather system details. In this specific case, the script aims to retrieve data from the registry and execute specific commands based on the privilege level under which the script operates.

Additionally, attackers may employ PowerShell commands such as "Get-GPPPassword.ps1" [75] to extract sensitive data from compromised systems, such as login credentials and personal information.

3 Architecture of Graphite

Figure 2 gives the architecture of Graphite, which is designed for fileless malware detection in enterprise network environments. In Graphite, logs are collected using SilkService of ETW and are forwarded to Logstash, where logs are parsed and transferred to ElasticSearch [63]. ElasticSearch then indexes and processes these logs to make them searchable and analyzable. We chose Elasticsearch as the log storage solution over traditional disk storage due to its scalability and rapid data storage and retrieval capability. Our decision is motivated by the substantial event loss experienced when storing ETW logs on disk, due to the considerably slower disk writing speed compared to the log generation speed [22]. By directing logs to ElasticSearch, we prevent event loss and enable real-time capture of log data.

Next, computation graphs are constructed from the ETW logs, abstracting log entities and their interactions from system events. Unlike provenance graphs, which are process-centered, our computation graphs are thread-centered graphs, which provides finer-grained information specific to each thread. Given the gigantic size of computation graphs, it is challenging to apply machine learning techniques to them directly. To address this, we develop a projection algorithm that decomposes each computation graph into smaller graphlets. These graphlets are then fed into a pre-trained graph-based model for malware detection.

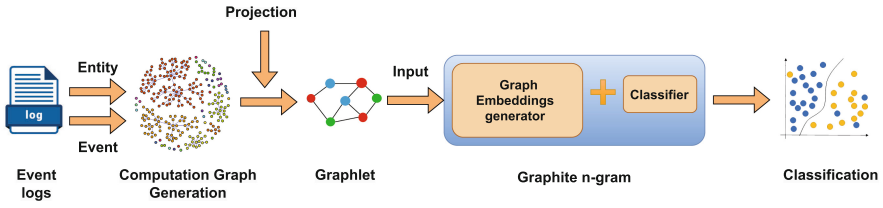


Fig. 2. Architecture of Graphite .

3.1 Log Schema

To facilitate the creation of computation graphs from ETW logs, we designed a schema to define entities and their associated attributes. The schema defines five types of entities, including two subjects (Thread and Process) and three objects (File, Registry, and Network).

The Thread entity serves as a central hub in our schema. As mentioned in Sect. 1, unlike other provenance graphs that are process centered [31,32,34,

Table 1. Node-level and edge-level attributes

Type of Node	Attribute Name	Attribute Description
Process Node Attributes	ProcessID	Parent process ID
File Node Attributes	FileName	Full file path
	FileObject	File handler
Registry Node Attributes	KeyObject	Registry key handler
	RelativeName	Registry key path
Network Node Attributes	daddr	Network destination address
Thread Node Attributes	XmlEventData_ThreadID	Child thread ID
	XmlEventData_ProcessID	Child process ID
	ThreadID	Parent thread ID
	ProcessID	Parent process ID
Process Edge Attributes	ImageName	Process image name
	XmlEventData_ThreadID	Child thread ID
	XmlEventData_ProcessID	Child process ID
	ThreadID	Parent thread ID
	ProcessID	Parent process ID
	CreateTime	Process creation time
	EventName	Event's string representation
	Opcode	Event's numeric representation
	TimeStamp	Event's logged time
File/Registry/Network Edge Attributes	XmlEventData_ThreadID	Child thread ID
	XmlEventData_ProcessID	Child process ID
	ThreadID	Parent thread ID
	ProcessID	Parent process ID
	EventName	Event's string representation
	Opcode	Event's numeric representation
	TimeStamp	Event's logged time

44,61,73], our thread-centered graphs provide finer-grained information specific to each thread. Our experimental results in Sect. 5 show that leveraging local temporal information associated with each thread improves the classification accuracy.

An event in the schema specifies a connection between two entities. For instance, an event can be a thread of a process reading a file or a thread transmitting data to a network socket. Process creation events connect two processes in the schema via thread, while thread creation events connect a process and a thread. In our schema, each entity is associated with one or more attributes, which specify whether the entity is a subject or an object, as well as its sub-type (e.g., File or Process). Events can also have attributes such as timestamp and event type. When creating the computation graph, we select persistent attributes as node attributes and non-persistent attributes as edge attributes. Persistent attributes, such as "FileName" do not change the identity of the node. Non-persistent attributes such as "TimeStamp" may change over time and alter the

node's state. The complete list of node attributes and edge attributes in our schema is given in Table 1.

3.2 Computation Graphs

Computation graphs are constructed based on the schema. Nodes in the computation graph represent log entities, and edges represent events. Table 2 lists the types of events that can occur between two specified types of entities.

Table 2. Event types

Edge types	List of events
File → Thread	DirEnum/DirNotify/Read/QueryEA/QuerySecurity/QueryInformation
Thread → File	Cleanup/Close/Create/CreateNewFile/DeletePath/ FSCTL/Flush/NameDelete/NameCreate/SetDelete/ SetInformation/OperationEnd/Rename/RenamePath/Write
Process → Thread	ThreadStart/ThreadStop/ThreadWorkOnBehalfUpdate
Thread → Process	CpuPriorityChange/CpuBasePriorityChange/IOPriorityChange/ ImageUnload/ImageLoad/JobStart/JobTerminate/ ProcessStart/ProcessStop/ProcessFreeze/PagePriorityChange
Network → Thread	Connectionaccepted/Datareceived/DatareceivedoverUDPprotocol/ Disconnectissued
Thread → Network	ConnectionAttempted/Dataretansmitted/DataSent/ DatasentoverUDPprotocol/Protocolcopieddataonbehalfofuser
Registry → Thread	35(QueryKey)/38(QueryValueKey)/39(EnumerateKey)/ 40(EnumerateValueKey)/41(QueryMultipleValueKey)/45(QuerySecurityKey)
Thread → Registry	32(CreateKey)/33(OpenKey)/34(DeleteKey)/36(SetValueKey)/ 37(DeleteValueKey)/42(SetInformationKey)/44(CloseKey)/ 46(SetSecuritykey)/13(RegPerfOpHiveFlushWroteLogFile)

Node and Edge events are denoted by "EventName" in the log entry except for registry nodes. "Opcodes" are used to identify events related to registry nodes, which are numeric representations of different events. For example, Opcode 35 corresponds to "QueryKey" [19], indicating that a Thread is querying key information from the registry. Edges in our computation graphs are directed, with the direction specifying the flow of information. For instance, if a process reads a file, an edge is constructed from the node representing the file to the node representing the process, indicating that information flows from the file to the process. Likewise, if a process sends data through the network, an edge is directed from the node representing the process to the node representing the network destination address. When a process starts a thread, an edge is created from the node representing the process to the node representing the thread.

We use a unique ID (UID) to uniquely identify each entity and event, which is a hash value computed using attributes of nodes and edges. Table 3 shows how

UID is computed. As a Process ID can be reused for different processes (e.g., after the termination of one process and the subsequent start of another), the UID of a process is computed using both the process ID and its corresponding creation time. To uniquely identify a Thread node, we use a combination of the thread ID, the associated Process's ID and creation time, and the timestamp of the "ThreadStart" event corresponding to the creation of the thread. For file identification, the UID of a file node is computed using the file name (FileName). Similarly, we use the relative path of a registry key (RelativeName) to uniquely identify it. IP address (daddr) is used to uniquely identify the network endpoint. Since each event occurs at a specific time, each edge is uniquely identified by the timestamp associated with the event. Additionally, we include the host name (HostName) in computing the UID to differentiate entities across different hosts.

Table 3. Node and edge UID rules

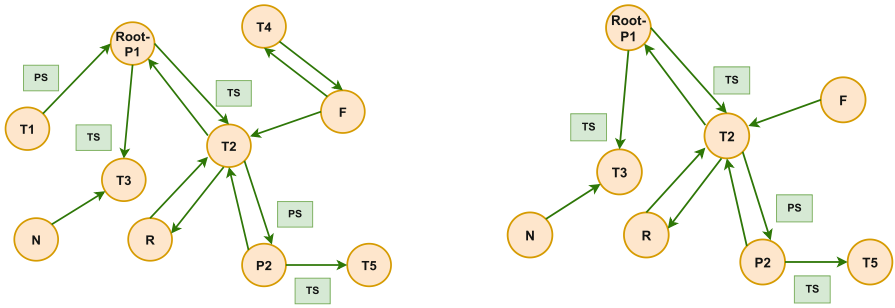
UID types	UID Rules
File Node	hash(FileName+HostName)
Process Node	hash(ProcessID+CreationTime+ HostName)
Registry Node	hash(RelativeName+HostName)
Network Node	hash(daddr + HostName)
Thread Node	hash(ProcessID+ThreadID+TimeStamp+CreationTime+HostName)
Edge	hash(ProcessID+ThreadID+EventName+Opcode+Timestamp+HostName)

To construct the computation graph from ETW logs, we first sort and parse event logs based on their respective timestamps. For each log entry, we create two nodes and a corresponding edge connecting these nodes. The computation graph is implemented using a graph library called igraph [70]. The resulting graph is a multi-graph, meaning that each event corresponds to one edge in the graph.

3.3 Graph Projection

Computation graphs contain rich information about process activities. However, such graphs are usually very large in enterprise network environments [62], making it difficult to apply machine learning techniques on them directly. To tackle this issue, we developed a process-based projection algorithm to decompose the computation graph into smaller graphlets.

Given a process ID, our projection algorithm first traverses the computation graph to locate the "ProcessStart" event that matches the process ID, which serves as the root of the projected graphlet. The algorithm then starts from the root process and uses a procedure similar to taint analysis [18] to prune the computation graph to include all and only the activities of a process, its descendant processes, and all threads associated with those processes. Descendant processes and associated thread nodes are identified by locating "ProcessStart" and



(a) The computation graph. (b) The graphlet.

Fig. 3. An example of graph projection with root process P_1 (TS: ThreadStart, PS: ProcessStart).

"ThreadStart" events during traversal from the root. File, registry, and network nodes linked to these processes/threads are also included in the graphlet. The complexity of our projection algorithm is $O(V + E)$, where V is the number of vertices and E is the number of edges in the graph.

Figure 3 illustrate the projection of a computation graph onto a graphlet. The root process in the graph is P_1. The graphlet contains all activities of P_1, its child process P_2, and its associated threads T_2, T_3, and T_5. File (F), network (N), and registry (R) nodes that are connected to these process/thread nodes are also added to the graphlet, which are leaf nodes of the graphlet.

4 Graph-Based Malware Classifications

We developed a graph-based approach to detect malware based on graphlets constructed from ETW logs. Our model, built upon Random Forest [10], leverages the temporal and spatial information within graphlets to perform malware classification. Random Forest was chosen as the base model due to its ability to capture complex and non-linear patterns in smaller datasets as well as its superior performance compared to alternative models such as logistic regression [45] and SVM [7].

Our graph-based approach leverages node types (e.g., process, file, registry, network, and thread) and edge-level attributes such as EventName and TimeStamp for malware detection. The temporal information utilized in Graphite is the chronological order of event timestamps, which can be classified as local or global. Local temporal information refers to the sorted sequence of event timestamps associated with specific graph components, such as a thread or between two nodes. Global temporal information refers to the chronological order of event timestamps across the entire graphlet. Spatial information includes any information derived from the connectivity relationships between nodes and edges such as information flow between neighboring nodes via edges or the leverage of local neighborhood specifics in graph-based methods.

Our graph-based approach is designed based on the computation graph schema described in Sect. 3.1, where threads serve as central entities for establishing connections with all other node types, while connections among other node types are restricted. As described in Sect. 3, thread-centered scheme provides finer-grained temporal and spacial information (which is specific to each thread), than the process-centered scheme. Our graph-based approach leverages local temporal information associated with each thread node using N-gram EventName features extracted from graphlets and utilize spatial information surrounding the thread node by counting the types of its neighboring nodes.

A special case is when $N = 1$, where temporal information is no longer captured. Instead, the distribution of EventNames associated with a thread node is obtained.

4.1 Generation of Graph Embeddings

This section describes how Graphite generates graph embeddings. We use an N-gram CountVectorizer [69] to learn and recognize N-gram EventName features from sorted EventName sequences of all threads from all training graphlets. For each thread node in a graphlet, its associated chronologically ordered EventName sequence is obtained from the EventName and TimeStamp attributes of its incoming and outgoing edges.

Next, we utilize the CountVectorizer to count the occurrences of N-gram EventName features within the sorted EventName sequence of each thread node.

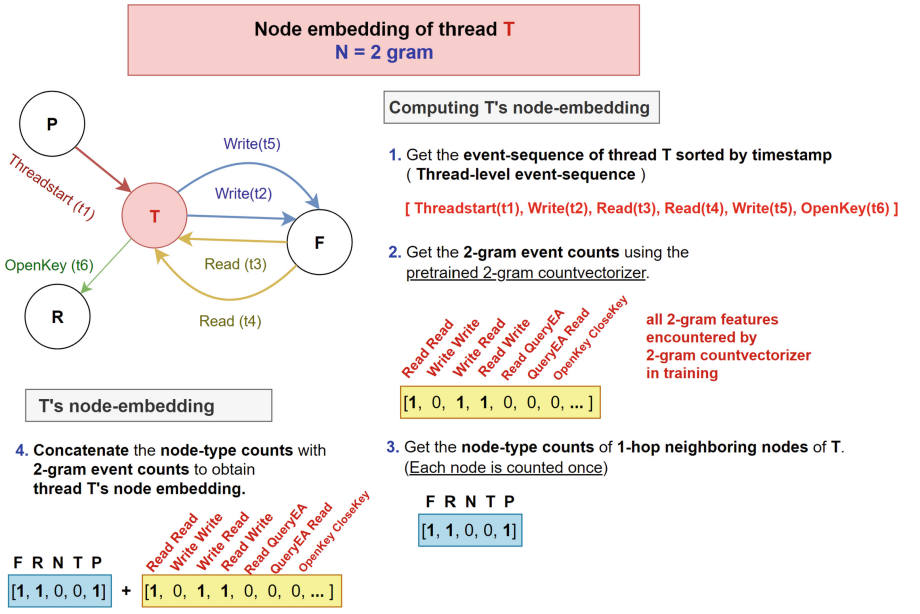


Fig. 4. Generation of thread node embeddings.

This produces an N-gram EventName feature vector, capturing temporal information local to each thread node. We also count the number of neighboring nodes of each type that are connected to each thread node by incoming or outgoing edges. We ensure no duplicate counts for the same neighbor, even with multiple edges in between. The alternative counting approach, allowing multiple counts for the same neighbor, provides interaction frequency and is used in our Extended N-gram approach described in Sect. 4.4. The thread node embedding is the concatenation of the thread node's N-gram EventName feature vector and its neighboring node-type count vector. Figure 4 provides an example illustrating how node embeddings are generated for a thread node. The graph embedding is derived using sum-pooling [79], aggregating all thread node embeddings using the element-wise vector-sum operation.

4.2 Data Encoding and Pre-processing

We utilize non-machine-specific attributes such as "EventName" for feature extraction. Because the "EventName" attribute comprises 59 known categorical values listed in Table 2, we use one-hot encoding to convert it into a binary vector of 0's and 1's. Two additional features are added to account for 'Null' and 'Unknown' events in ETW logs. This process resulted in a bit-vector representation with a length of 61 bits. The "Timestamp" attribute, converted into a scalar from its original date-time format, is utilized alongside the "EventName" feature vector to compose edge features. For node features, we utilize a 5-bit vector where each bit denotes a distinct node type, namely file, registry, network, process, and thread. The resulting node and edge features are then fed into the downstream machine learning model.

4.3 Real-Time Malware Detection

The log collection and graphlet generation for our real-time malware detection follow the procedure described in Sect. 3. However, waiting for each process to complete its execution before classification is impractical for real-time detection. To address this, we perform classification for a process when one of the following conditions holds: (1) the process completes execution, (2) the size of graphlet reaches a pre-defined threshold determined by malware analysts, or (3) the process remains inactive for an extended period. Moreover, graphlet generation and malware classification can run in parallel across multiple GPUs/CPUs, allowing to produce classification results in real-time. Performance results of our real-time detection are given in Sect. 5.3.

4.4 Alternative Graph-Based Approaches

We also experimented with graph-based approaches outlined below, whose detection accuracy is presented in Sect. 5.

Extended N-gram extends Graphite N-gram by utilizing additional features that provide spatial information around thread nodes. Such features include the

Table 4. Comparison of different graph-based approaches.

Graph-based Approaches	Spatial Information	Temporal Information
Graphite N-gram	Local to thread	Local to thread
Extended N-gram	Local to thread	Local to thread
Standard K-hop message passing	Information flow	None
N-gram K-hop message passing	Information flow	Local to edge

number of times a thread node interacts with its neighboring nodes and the average number of thread nodes to which a file, registry, network, and process node connects.

Standard K-hop message passing updates node embeddings by aggregating information, such as EventName of incoming edges and node types, from K-hop neighbors of nodes (i.e., nodes that are reachable within K edges) via K iterations [14]. This approach utilizes spatial information from graphlets, but does not incorporate temporal information.

N-gram K-hop message passing integrates temporal information into the standard $K-$hop message passing by incorporating N-gram EventName features extracted from the event sequence sorted by timestamp of incoming edges. The extraction of N-gram EventName features is facilitated by the conversion of the graphlet, a multigraph with each edge representing an event, into timestamp-sorted event sequences on edges.

Table 4 summarizes the temporal and space information utilized in different graph-based approaches.

Table 5. Node and edge statistics of malware and benign graphlets.

	Malware		Benign	
	Mean	Standard Dev.	Mean	Standard Dev.
Node count	1440.58	803.95	1363.89	624.56
Edge count	18577.22	14275.37	18066.90	14459.97
File node	597.03	480.55	593.26	379.07
Network node	1.26	3.67	1.12	0.63
Registry node	786.63	521.53	734.92	458.26
Process node	2.11	2.61	1.96	6.78
Thread node	53.54	296.35	32.63	61.34

5 Experimental Evaluation

This section presents our experimental results. Three machines were used to train and tune all models. One machine is equipped with a 2.65-3.6 GHz AMD

EPYC 7413 CPU and NVIDIA RTX A6000 48GB GPUs, another has a 2.8–3.35 GHz AMD EPYC 7402P CPU and NVIDIA Tesla T4 15GB GPUs, and the third is a 2.90–3.50 GHz machine with Intel Xeon Gold 6326 processor.

5.1 Data Collection

We collected malicious and benign PowerShell scripts from VirusTotal [1] and over 15 other websites [5,13,20,21,30,33,42,49–53,55,56,59,67,68,71,75]. Each PowerShell script labelled as malicious has been confirmed as malicious by multiple malware detection engines on VirusTotal. The collected PowerShell scripts exhibit diverse writing styles. Some scripts took a considerable amount of time to run due to a large number of cmdlets and complex looping structures, while others finished execution more quickly. Additionally, many malicious PowerShell scripts require appropriate parameters, keyboard input, or a specific software version. We analyzed and executed 1,152 benign and 949 malware scripts downloaded from the above websites and excluded scripts that failed to execute or could not complete their major functionalities. The final dataset used in our experiments comprises 690 malicious PowerShell scripts and 771 benign ones. We collected logs using four ETW event providers [78]: (1) *Microsoft-Windows-Kernel-Process* records operations on processes and threads; (2) *Microsoft-Windows-Kernel-File* captures file-related activities; (3) *Microsoft-Windows-Kernel-Registry* records Windows registry operations; and (4) *Microsoft-Windows-Kernel-Network* records network-related activities. To execute malicious PowerShell scripts, we established an isolated virtual machine (VM) with Internet connections, ensuring that any activities or changes within the isolated virtual machine do not impact the host system or other virtual machines. To prevent machine learning models from performing classification based on machine/VM-specific information, we collected benign and malicious log samples using the same VM configuration. The total number of log entries is 81,726,548 and 96,287,768 for malware and benign samples, respectively.

Table 5 gives the average number of nodes and edges in graphlets and their standard deviations. The table shows that, on average, malware graphlets contain slightly more nodes and edges than benign graphlets. Moreover, the standard deviation of node counts in malware graphlets is higher than that in benign ones. There is no apparent linear relationship to determine whether a graphlet is benign or malicious based solely on the number of nodes or edges. The table also shows that malicious graphlets tend to have more thread nodes than benign ones. While mean values differ between malware and benign samples across different feature types, the substantial overlap in standard deviations suggests that individual feature types are not highly discriminatory for distinguishing between malware and benign samples.

5.2 Effectiveness of Graphite

This section evaluates the effectiveness of Graphite against baseline models that ignore the graph structure. The baseline models are Random Forest with 1, 2,

and 4-gram features. In the 1-gram model, the model simply counts the number of node types and event names as features, leveraging neither temporal nor spatial information of graphlets. The 2-gram and 4-gram models utilize N-gram EventName features extracted from the global timestamp-ordered sequence of events (as opposed to per-thread event sequences), combined with node type counts. These models leverage global temporal information of graphlets, but not spatial information.

Table 6. Comparison of average validation scores and classification accuracy.

	Mean Avg. Val.Acc.	Mean Avg. Val.F1	Max Avg. Val.Acc.	Max Avg. Val.F1	Test Accuracy	Test F1-Score
Baseline 1-**gram**	0.749	0.715	0.791	0.772	0.784	0.767
Baseline 2-**gram**	0.739	0.689	0.808	0.788	0.791	0.776
Baseline 4-**gram**	0.780	0.715	0.868	0.853	0.849	0.832
Graphite 1-**gram**	0.748	0.714	0.791	0.771	0.781	0.768
Graphite 2-**gram**	0.762	0.717	0.833	0.814	0.815	0.797
Graphite 4-**gram**	**0.817**	**0.774**	**0.898**	**0.887**	**0.877**	**0.863**
Extended 4-**gram**	**0.815**	**0.772**	**0.896**	**0.885**	**0.883**	**0.872**
Standard 3-**hop M.P.**	0.719	0.674	0.772	0.745	0.767	0.736
4-gram 3-**hop M.P.**	0.720	0.677	0.772	0.745	0.781	0.764
Combined 4-**gram**	0.804	0.752	0.893	0.882	0.870	0.858
FEATHER-GRAPH	0.642	0.596	0.670	0.634	0.648	0.619
Graph2Vec	0.814	0.800	0.871	0.862	0.600	0.580
GAT	0.706	0.664	0.761	0.731	0.6007	0.5339
GIN	0.710	0.710	0.786	0.771	0.604	0.567

We used 80% of the dataset for training and the remainder for testing. To ensure a balanced representation, we perform a stratified split [38] on the dataset, not only by the label but also by the source of PowerShell scripts (i.e., the websites from which the scripts were obtained). This approach is employed because scripts originating from the same source tend to exhibit common attributes such as writing styles, which may not accurately reflect the script's actual behavior. Performing a stratified split mitigates potential biases introduced by an imbalanced distribution of data sources in the dataset.

The model-tuning process involved an extensive exploration of hyperparameter ranges, comprising 6912 sets generated by combining typical values for hyperparameters. We employed k-fold cross-validation with $k = 10$, a widely accepted choice. This approach avoids reliance on a single hyperparameter configuration, such as default settings, which is crucial for preventing "unintentional hyperparameter hacking" [24]. Hyperparameters yielding the highest validation scores were utilized to assess the model's classification accuracy and F1 scores during testing.

Table 6 presents the mean and maximum average validation scores, test accuracy and test F1-scores of the baseline approach, Graphite and other graph-based

approaches discussed in Sect. 4.4, unsupervised graph-embedding approaches Graph2Vec [48] and FEATHER-GRAPH [58], and two conventional Graph Neural Network (GNN) models GAT [72] and GIN [76]. The mean and the maximum average validation scores were obtained from hyperparameter tuning based on 10-fold Cross Validation, which represent the overall and best performance scores across various hyperparameter sets and splits within the training set, respectively. The test scores reflect the performance of the best model on the test set, corresponding to the hyperparameter set with the maximum average validation score.

For both baseline models and Graphite, we conducted experiments with 1, 2, and 4-gram features. We did not choose higher values, as extracting N-gram features from the long global event sequence could result in significant high feature dimensions. In the Standard message passing approach, we experimented with 3 hops to ensure sufficient information flow. As 4-gram models consistently do better than 1-gram and 2-gram models, we obtained the performance results of 4-gram for Extended N-gram and N-gram k-hop message passing approach. The Combined N-gram approach in the table concatenates the graph embeddings of Graphite 4-gram and Baseline 4-gram. This integration aims to provide the model with both local and global temporal information, exploring potential complementary effects.

To evaluate the performance of FEATHER-GRAPH [57], we followed the experimental setup outlined in the original paper for graph classification, including the hyperparameter settings. This setup utilizes the topological feature of node degree to derive graph embeddings, without using any trainable parameters. Graph2Vec is a neural embedding model that includes neural hyperparameters such as the graph embedding dimension and the number of learning epochs, along with other non-neural hyperparameters. We used the best empirical hyperparameter combination that yielded the best performance in the downstream task to evaluate the implementation of Graph2Vec given at [57]. Once we obtained graph embeddings from FEATHER-GRAPH and Graph2Vec, we used Random Forest as the classifier and fine-tuned it using the same hyper-parameter combinations as Graphite. GAT and GIN were fine-tuned with hyperparameter combinations that focused on lower model complexity configurations to avoid overfitting on small datasets.

We observe that Graphite 4-gram achieves the highest average validation score and classification accuracy, indicating that local temporal information is most effective in distinguishing between benign and malware samples. The Extended 4-gram has similar validation scores and accuracy, suggesting that the additional spatial features have minimal impact. Among baseline models, the 4-gram model performs the best, followed by the 2-gram model, suggesting that global temporal information also helps improve the classification accuracy. With 1-grams, Graphite demonstrates comparable performance to the baseline model, suggesting that neighboring node-type information for thread nodes has minimal effect on performance.

The table also shows that both baseline and Graphite exhibit lower classification accuracy compared to the maximum average validation score for 1-gram and 2-gram. The classification accuracy of Graphite 4-gram is similar to its maximum average validation score. Additionally, Graphite outperforms its baseline counterparts in 2-gram and 4-gram, achieving 4% higher test accuracy and test F1 score.

Moreover, the table shows that message passing performs worse than the baseline models. As threads serve as the central entities in our schema, it is common to observe many cycles between threads and other node types within the computation graph. In our dataset, over 50% of distinct node pairs exhibit cyclic relationships for both benign and malware graphlets. In such a topology, multi-hop information propagation leads to an excess of cyclic information flows, a recognized issue with message passing [14]. The conventional GNN models, GAT and GIN, which leverage message passing to extract spatial information, also did not perform well. This can be attributed to GNNs being prone to over-smoothing [16] in such highly cyclic topology.

The table also shows that FEATHER-GRAPH did not perform well. This is because the graph embeddings generated from FEATHER-GRAPH captures the structural information of the graph, but benign and malware graphlets are not easily distinguishable based on graph structure. This is based on the observation that in both benign and malware graphlets, there are typically 4−5 active threads interacting with different node types, with a higher number of connections linked to file and registry nodes. Graph2Vec achieves high average validation scores but the lowest test scores. Graph2Vec trains an embedding neural network to produce similar embeddings for structurally similar graphs. It captures nuanced structural information within the training graphs, leading to high validation scores. However, the low test scores indicate that Graph2Vec may not generalize well to unseen graphs. Some researchers suggest that Graph2Vec is non-inductive and does not naturally generalize to unseen graphs outside the training set [4].

5.3 Real-Time Responsiveness

It took 10 seconds to start Silkservice and 45 seconds to initialize the logstash. Logstash processes and transfers event logs to Elasticsearch without any noticeable delay. On average, it takes approximately 3.07 seconds to collect 1,000 log entries. Generating one graphlet from 1,000 log entries takes approximately 0.9 seconds on average, including the time to construct the computation graph (0.83 seconds) and perform the projection to generate the graphlet (0.06 seconds). On average, producing prediction results for samples with 1,000 nodes takes approximately 0.08 seconds for Graphite 2-gram and 0.56 seconds for 4-gram, on average.

Table 7 presents the classification accuracy and F1-score of baseline models and Graphite using threshold values of 500 and 1,000 nodes, which correspond to roughly one-third and two-thirds of the average number of nodes in graphlets, respectively. The table shows that, for both thresholds, the accuracy of F-1 score of Graphite 1-gram and 2-gram is similar to their baseline counterparts. The

Table 7. Real-Time detection accuracy with a threshold of 500 and 1,000 nodes.

	Test Accuracy #500	Test F1-Score #500	Test Accuracy #1000	Test F1-Score #1000
Baseline 1-**gram**	0.670	0.686	0.741	0.725
Baseline 2-**gram**	0.758	0.734	0.741	0.723
Baseline 4-**gram**	0.802	0.779	0.808	0.799
Graphite 1-**gram**	0.706	0.697	0.747	0.736
Graphite 2-**gram**	0.730	0.713	0.758	0.744
Graphite 4-**gram**	**0.846**	**0.829**	**0.867**	**0.859**

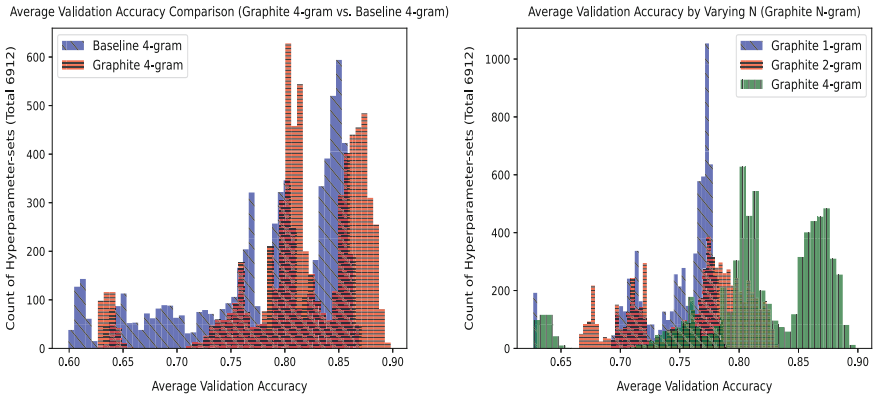
classification accuracy of Graphite 4-gram is the best for both 500 and 1,000 thresholds, which is close to that of offline detection (0.846 vs. 0.877 and 0.867 vs. 0.877, respectively), indicating that Graphite 4-gram performs well even with partial sub-graphs.

Additionally, Graphite consume 135 MB memory for classifying each sample, which includes the memory used for loading one graphlet into memory, extracting the feature vector from that graphlet, and making a prediction.

5.4 Distribution of Average Validation Scores

Figure 5(a) presents the distribution of average validation scores of Graphite 4-gram and Baseline 4-gram across all 6912 hyperparameter sets evaluated using 10-fold cross validation. The figure shows that the average validation scores of Graphite 4-gram are consistently higher than those of baseline 4-gram, indicating the sustained effectiveness of Graphite 4-gram rather than being limited to isolated outliers. Figure 5(b) shows that higher N in N-gram features improves the validation accuracy.

The figure also visualizes the distribution's shape, which is required information in selecting the appropriate hypothesis testing method. Given the paired nature of the average validation scores between Graphite 4-gram and baseline 4-gram, and considering their non-normal, multimodal distributions observed in the figure, we selected a paired non-parametric permutation test with median test statistic to assess the statistical significance of the score differences. Utilizing the median test statistic to compare central tendencies, 1-gram results (p-values of 0.9804 for average validation accuracy and 0.29 for average validation F1-scores) indicated no significant difference. However, for both 2-gram and 4-gram, p-values of 0.0002 for average validation scores demonstrated statistical significance.



(a) Graphite 4-gram vs. Baseline 4-gram (b) Graphite N-gram by varying N

Fig. 5. Average validation accuracy distribution.

6 Related Work

This section presents related work on fileless malware attack detection and graph-based malware detection.

6.1 Fileless Malware Attacks

Memory-based detection methods examine computer memory to unveil traces left by fileless malware attacks [9,36,37,64,80]. These techniques extract features from memory contents and then apply rule-based methods or machine learning models to detect fileless malware. However, most of these techniques rely on known malware signatures for identification, and signature-based detection methods are prone to evasion attacks [60]. Additionally, these techniques can cause significant performance degradation due to continuous memory scanning [8].

Static analysis techniques have been developed to detect malicious PowerShell scripts. Mimura et al. [46] decomposed PowerShell scripts into individual words and utilized them as machine learning features to detect fileless malware. Danny et al. [27] detected evasive malicious PowerShell commands using natural language processing and character-level Convolutional Neural Networks (CNNs). Choi et al. [17] leveraged Graph Convolutional Networks (GCNs) to calculate Jaccard similarities between new and existing scripts and used these similarities to construct an adjacency matrix for the detection of malicious activities. Bucevschi et al. proposed techniques based on the perceptron algorithm [54] to detect anomalies in PowerShell code by extracting relevant features, and building a model to describe a malicious command [11]. Li et al. [43] developed a sub-tree based deobfuscation method to help detect malicious obfuscated PowerShell scripts. However, the above approaches require the source code of the

scripts and may be thwarted by dynamic script generation and sophisticated code obfuscation.

Researchers have also employed dynamic analysis for malware detection. Lanzi et al. [41] proposed to use a system-centric access activity model, which monitors interactions between benign programs and the operating system, to detect malware. This approach, however, detects only malware that tampers with binaries or settings of the operating system or applications. Jindal et al. [35] proposed Neurlux, which utilizes a Cuckoo sandbox to generate dynamic analysis reports detailing behavioral information, and uses word sequences present in these reports to predict whether a report is from a malicious binary. However, this work focuses on classifying given samples rather than detecting malware in real time.

6.2 Graph/Provenance-Based Malware Detection

There have been numerous attempts at detecting malware attacks using provenance-based techniques. These techniques involve the construction of provenance graphs from computer system logs, which are further examined to identify malware attacks. Shu et al. [62] identified malicious behavior by constructing computation graphs that support historical and real-time threat detection. SLEUTH [31] performs policy-based real-time attack detection and reconstruction using computation graphs constructed from system call logs. [61] integrated audit logs and taint analysis to trace historical provenance data and detect anomalous activities within the system. In [44], the authors developed malware detection techniques through analysis of ETW logs and critical sections of application executables. RAIN [34] records system-call events at runtime and conducts dynamic information flow tracking (DIFT) to detect attacks. Wang et al. [32] employed system audit logs and dependency-preserving log reduction methods to offer insights into attack specifics (e.g., the direction, timing and the execution details) of Advanced Persistent Threats (APTs). The above works focus on rule-based approaches to detect malicious activities. Our method, in contrast, applies graph-based machine learning methods for automated detection of fileless malware. Wang et al. [73] proposed ProvDetector, a provenance-based approach for detecting stealthy malware. This method uses a set of benign provenance graphs generated from the same program in various environment as the training dataset to build a model to detect if the program has been hijacked by stealthy malware. However, ProvDetector cannot detect malicious software other than the monitored programs. UNICORN [25] constructs provenance graphs using publicly available datasets and employs a K-medoids clustering algorithm to detect anomalies related to APT attacks that deviate from the host's normal evolving behavior. Unlike our approach which focuses on detecting fileless malware, UNICORN is specifically tailored to the characteristics of APTs. Additionally, the provenance graphs proposed in the aforementioned work are process-centered. In contrast, our computation graphs are thread-centered, providing finer-grained information specific to each thread.

Despite the success of graph neural networks (GNNs) in other graph-based domains, their utilization in malware detection is relatively limited. Yan et al. [77] used Deep Graph Convolutional Neural Networks (DGCNNs) [79] to classify malware executables represented as CFGs. SDGNet [81] uses a Graph Convolutional Network (GCN) to classify malware samples based on their control flow graphs. DL-FHMC [2] utilizes CFG-based behavioral patterns for adversarial IoT malicious software detection. Soteria [3] employs an auto-encoder and a CNN architecture to detect and classify malware samples represented as CFGs. Kang et al. [39] applied static analysis techniques to construct function call graphs (FCGs) from malware programs and utilizes an ensemble classifier to detect malware based on FCGs. Herath et.al. [29] propose techniques to identify subgraphs of the malware CFG that contribute most towards classification and provide insight into importance of the nodes within it. However, as the above works require static analysis of the malware programs to construct FCGs or CFGs, they are not suitable for malware detection in situations where these malware programs are not available, such as fileless malware attacks.

7 Conclusion and Future Work

This paper presents Graphite, a graph-based approach to automatic detection of fileless malware attacks in real-time. Graphite generates computation graphs from system event logs collected via ETW and projects them into smaller graphlets, which are then fed into a graph-based malware detection model. We have evaluated the effectiveness of Graphite using benign and malicious PowerShell scripts from numerous sources. Our experimental results show that Graphite achieves 87.7% classification accuracy in offline testing, and 84.6% and 86.7% accuracy in real-time detection for two different pre-defined thresholds. In the future, we plan to leverage explainable AI to further improve the detection accuracy of Graphite. We also plan to develop methods to bridge the gap between explanations generated by explainable AI tools and those that human analysts can easily understand.

Acknowledgement. This work is supported in part by a SUNY-IBM AI Research Alliance grant. We thank the anonymous reviewers for their constructive comments.

References

1. Virustotal (2023). https://www.virustotal.com
2. Abusnaina, A., et al.: DL-FHMC: deep learning-based fine-grained hierarchical learning approach for robust malware classification. IEEE Trans. Dependable Secure Comput. **19**(5), 3432–3447 (2021)
3. Alasmary, H., et al.: Soteria: detecting adversarial examples in control flow graph-based malware classifiers. In: International Conference on Distributed Computing Systems, pp. 888–898 (2020)

4. Bai, Y., et al.: Unsupervised inductive graph-level representation learning via graph-graph proximity. In: International Joint Conference on Artificial Intelligence, pp. 1988–1994. ijcai.org (2019)
5. BlackSnufkin: pt-toolkit (2022). https://github.com/BlackSnufkin/PT-ToolKit/tree/f78567ce9b4701acfd6af21196b95eef44bbc9c5/PowerShell-Scripts
6. Bohannon, D.: Invoke-obfuscation (2019). https://github.com/danielbohannon/Invoke-Obfuscation
7. Boser, B.E., Guyon, I.M., Vapnik, V.N.: A training algorithm for optimal margin classifiers. In: Workshop on Computational Learning Theory, pp. 144–152 (1992)
8. Botacin, M., Grégio, A., Alves, M.A.Z.: Near-memory & in-memory detection of fileless malware. In: Symposium on Memory Systems, pp. 23–38 (2020)
9. Bozkir, A.S., Tahillioglu, E., Aydos, M., Kara, I.: Catch them alive: a malware detection approach through memory forensics, manifold learning and computer vision. Comput. Sec. **103**, 102166 (2021)
10. Breiman, L.: Random forests. Mach. Learn. **45**(1), 5–32 (2001)
11. Bucevschi, A.G., Balan, G., Prelipcean, D.B.: Preventing file-less attacks with machine learning techniques. In: 2019 21st International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), pp. 248–252. IEEE (2019)
12. Buckbee, M.: What is fileless malware? PowerShell exploited (2023). https://www.varonis.com/blog/fileless-malware
13. Camichel, C.: Bazaar by abuse.ch (2024). https://bazaar.abuse.ch/browse.php?search=tag%3Aps1
14. Cantwell, G.T., Newman, M.E.: Message passing on networks with loops. Proc. Natl. Acad. Sci. **116**(47), 23398–23403 (2019)
15. Chai, H., Ying, L., Duan, H., Zha, D.: Invoke-deobfuscation: Ast-based and semantics-preserving deobfuscation for powershell scripts. In: International Conference on Dependable Systems and Networks, pp. 295–306 (2022)
16. Chen, D., Lin, Y., Li, W., Li, P., Zhou, J., Sun, X.: Measuring and relieving the over-smoothing problem for graph neural networks from the topological view. In: AAAI Conference on Artificial Intelligence, vol. 34, pp. 3438–3445 (2020)
17. Choi, S.: Malicious PowerShell detection using graph convolution network. Appl. Sci. **11**(14), 6429 (2021)
18. Clause, J., Li, W., Orso, A.: Dytan: a generic dynamic taint analysis framework. In: International Symposium on Software Testing and Analysis, pp. 196–206 (2007)
19. Damari, O.: (2019). https://github.com/repnz/etw-providers-docs/blob/master/Manifests-Win10-10240/Microsoft-Windows-Kernel-Registry.xml
20. Fleschutz, M.: Powershell (2023). https://github.com/fleschutz/PowerShell/tree/main/scripts
21. Godoy, E.G.: Sysadmin-survival-kit-scripts (2021). https://github.com/ErickRock/Sysadmin-Survival-Kit-Scripts
22. Gwak, J.Y., Wakodikar, P., Wang, M., Yan, G., Shu, X., Stoller, S.D., Yang, P.: Debugging malware classification models based on event logs with explainable ai. In: 2023 IEEE International Conference on Data Mining Workshops (ICDMW), pp. 939–948 (2023)
23. Gómez, A.M.M., Bot, C., Mahmoud, M., Chernofsky, E.: (2019). https://github.com/mandiant/SilkETW
24. Hamilton, W.L., Ying, Z., Leskovec, J.: Inductive representation learning on large graphs. In: Annual Conference on Neural Information Processing Systems (NIPS 2017), pp. 1024–1034 (2017)

25. Han, X., Pasquier, T., Bates, A., Mickens, J., Seltzer, M.: Unicorn: Runtime provenance-based detector for advanced persistent threats. arXiv preprint arXiv:2001.01525 (2020)
26. Help Net Security: Fileless attacks increase 1,400% (2023). https://www.helpnetsecurity.com/2023/07/04/threat-actors-detection-evasion/
27. Hendler, D., Kels, S., Rubin, A.: Detecting malicious powershell commands using deep neural networks. In: Asia Conference on Computer and Communications Security, pp. 187–197 (2018)
28. Hendler, D., Kels, S., Rubin, A.: AMSI-based detection of malicious PowerShell code using contextual embeddings. In: ACM Asia Conference on Computer and Communications Security, pp. 679–693 (2020)
29. Herath, J.D., Wakodikar, P.P., Yang, P., Yan, G.: Cfgexplainer: explaining graph neural network-based malware classification from control flow graphs. In: 2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 172–184 (2022)
30. Hochwald, J.: Powershell-collection (2023). https://github.com/jhochwald/PowerShell-collection
31. Hossain, M.N., et al.: SLEUTH: Real-time attack scenario reconstruction from COTS audit data. In: USENIX Security Symposium, pp. 487–504 (2017)
32. Hossain, M.N., Wang, J., Sekar, R., Stoller, S.D.: Dependence-Preserving data compaction for scalable forensic analysis. In: USENIX Security Symposium, pp. 1723–1740 (2018)
33. IAMinZoho: Offsec-powershell (May 2023). https://github.com/IAMinZoho/OFFSEC-PowerShell
34. Ji, Y., et al.: Rain: refinable attack investigation with on-demand inter-process information flow tracking. In: Computer and Communications Security (CCS), pp. 377–390 (2017)
35. Jindal, C., Salls, C., Aghakhani, H., Long, K., Kruegel, C., Vigna, G.: Neurlux: dynamic malware analysis without feature engineering. In: Proceedings of the 35th Annual Computer Security Applications Conference, pp. 444–455 (2019)
36. Kara, I.: Fileless malware threats: recent advances, analysis approach through memory forensics and research challenges. Expert Syst. Appli., 119133 (2022)
37. Khalid, O., et al.: An insight into the machine-learning-based fileless malware detection. Sensors **23**(2), 612 (2023)
38. Kohavi, R.: A study of cross-validation and bootstrap for accuracy estimation and model selection. In:14th International Joint Conference on Artificial Intelligence, vol. 14(2) (2001)
39. Kong, D., Yan, G.: Discriminant malware distance learning on structural information for automated malware classification. In: ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 1357–1365 (2013)
40. Kumar, S., et al.: An emerging threat fileless malware: a survey and research challenges. Cybersecurity **3**(1), 1–12 (2020)
41. Lanzi, A., Balzarotti, D., Kruegel, C., Christodorescu, M., Kirda, E.: Accessminer: using system-centric models for malware protection. In: Proceedings of the 17th ACM Conference on Computer and Communications Security, pp. 399–412 (2010)
42. Larrubia, P.: Win-debloat-tools (2023). https://github.com/LeDragoX/Win-Debloat-Tools/tree/main/src/scripts
43. Li, Z., Chen, Q.A., Xiong, C., Chen, Y., Zhu, T., Yang, H.: Effective and lightweight deobfuscation and semantic-aware attack detection for powershell scripts. In: Computer and Communications Security, pp. 1831–1847 (2019)

44. Ma, S., Lee, K.H., Kim, C.H., Rhee, J., Zhang, X., Xu, D.: Accurate, low cost and instrumentation-free security audit logging for windows. In: Annual Computer Security Applications Conference, pp. 401–410 (2015)
45. Maalouf, M.: Logistic regression in data analysis: an overview. Inter. J. Data Analy. Techniq. Strategies **3**(3), 281–299 (2011)
46. Mimura, M., Tajiri, Y.: Static detection of malicious PowerShell based on word embeddings. Internet of Things **15**, 100404 (2021)
47. Mittal, N.: nishang (2023). https://github.com/samratashok/nishang
48. Narayanan, A., Chandramohan, M., Venkatesan, R., Chen, L., Liu, Y., Jaiswal, S.: graph2vec: Learning distributed representations of graphs. arXiv preprint arXiv:1707.05005 (2017)
49. Nefedov, D.: (2023). https://github.com/farag2/Utilities
50. Olin, B.: Terminal-icons (2023). https://github.com/devblackops/Terminal-Icons
51. RaouzRouik: smallposh (2022). https://github.com/RaouzRouik/smallposh
52. RiskyDissonance: poshc2 (2022). https://github.com/nettitude/PoshC2
53. Rodriguez, N.: Powershell-scripts (2023). https://github.com/nickrod518/PowerShell-Scripts
54. Rosenblatt, F.: The perceptron: a probabilistic model for information storage and organization in the brain. Psychol. Rev. **65**(6), 386 (1958)
55. Ross, C.: Randomps-scripts (2017). https://github.com/xorrior/RandomPS-Scripts
56. Ross, C.: Empire (2019). https://github.com/EmpireProject/Empire
57. Rozemberczki, B., Kiss, O., Sarkar, R.: Karate club: an api oriented open-source python framework for unsupervised learning on graphs. In: Proceedings of the 29th ACM International Conference on Information & Knowledge Management, pp. 3125–3132 (2020)
58. Rozemberczki, B., Sarkar, R.: Characteristic functions on graphs: birds of a feather, from statistical descriptors to parametric models. In: Proceedings of the 29th ACM International Conference on Information & Knowledge Management, pp. 1325–1334 (2020)
59. Russell, J.: powershell-scripts (2021). https://github.com/jrussellfreelance/powershell-scripts
60. Saad, S., Mahmood, F., Briguglio, W., Elmiligi, H.: Jsless: a tale of a fileless javascript memory-resident malware. In: 15th International Conference on Information Security Practice and Experience, pp. 113–131 (2019)
61. Shiqing, M., Zhang, X., Xu, D.: ProTracer: towards practical provenance tracing by alternating between logging and tainting. In: Network and Distributed System Security Symposium (2016)
62. Shu, X., et al.: Threat intelligence computing. In: ACM SIGSAC Conference on Computer and Communications Security, pp. 1883–1898 (2018)
63. Shukla, P., Kumar, S.: Learning Elastic Stack 7. 0 : distributed search, analytics, and visualization using elasticsearch, logstash, beats, and kibana. Packt Publishing (2019)
64. Sihwail, R., Omar, K., Arifin, K.A.Z.: An effective memory analysis for malware detection and classification. Comput. Mater. Continua **67**(2) (2021)
65. SonicWall: 2022 SonicWall cyber threat report (2022). https://sonicguard.com/datasheets/2022-sonicwall-cyber-threat-report.pdf
66. SonicWall: Mid-year update: 2022 SonicWall cyber threat report (2022). https://www.sonicwall.com/medialibrary/en/white-paper/mid-year-2022-cyber-threat-report.pdf

67. stevencohn: Windowspowershell (2023). https://github.com/stevencohn/WindowsPowerShell
68. Sutherland, S.: Powershellery (2023). https://github.com/nullbind/Powershellery
69. scikit-learn development team: Countvectorizer (2024). https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html
70. igraph core team, T.: Igraph (2003). https://igraph.org/python/tutorial/0.9.8/tutorial.html
71. Tworek, G.: Psbits (2023). https://github.com/gtworek/PSBits
72. Velickovic, P., Cucurull, G., Casanova, A., Romero, A., Lio', P., Bengio, Y.: Graph attention networks. ArXiv abs/ arXiv: 1710.10903 (2018)
73. Wang, Q., et al.: You are what you do: Hunting stealthy malware via data provenance analysis. In: NDSS (2020)
74. Warnars, N.: Detecting fileless malicious behaviour of .net c2 agents using etw (2020)
75. Will: Powersploit (2020). https://github.com/PowerShellMafia/PowerSploit
76. Xu, K., Hu, W., Leskovec, J., Jegelka, S.: How powerful are graph neural networks? In: International Conference on Learning Representations (2019)
77. Yan, J., Yan, G., Jin, D.: Classifying malware represented as control flow graphs using deep graph convolutional neural network. In: International Conference on Dependable Systems and Networks, pp. 52–63 (2019)
78. Yoshizaki, I.: Providers (2022). https://gist.github.com/guitarrapc/35a94b908bad677a7310
79. Zhang, M., Cui, Z., Neumann, M., Chen, Y.: An end-to-end deep learning architecture for graph classification. In: AAAI Conference on Artificial Intelligence, vol. 32 (2018)
80. Zhang, S., Hu, C., Wang, L., Mihaljevic, M.J., Xu, S., Lan, T.: A malware detection approach based on deep learning and memory forensics. Symmetry **15**(3), 758 (2023)
81. Zhang, Z., Li, Y., Dong, H., Gao, H., Jin, Y., Wang, W.: Spectral-based directed graph network for malware detection. IEEE Trans. Netw. Sci. Eng. **8**(2), 957–970 (2020)