Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

Malware classification through Abstract Syntax Trees and L-moments

Anthony J. Rose [*], Christine M. Schubert Kabban, Scott R. Graham, Wayne C. Henry, Christopher M. Rondeau

Air Force Institute of Technology, 2950 Hobson Way, WPAFB, 45433, OH, USA

ARTICLE INFO

Keywords:
Malware
Classification
Abstract Syntax Tree
L-moments
Network analysis

ABSTRACT

The ongoing evolution of malware presents a formidable challenge to cybersecurity: identifying unknown threats. Traditional detection methods, such as signatures and various forms of static analysis, inherently lag behind these evolving threats. This research introduces a novel approach to malware detection by leveraging the robust statistical capabilities of L-moments and the structural insights provided by Abstract Syntax Trees (ASTs) and applying them to PowerShell. L-moments, recognized for their resilience to outliers and adaptability to diverse distributional shapes, are extracted from network analysis measures like degree centrality, betweenness centrality, and closeness centrality of ASTs. These measures provide a detailed structural representation of code, enabling a deeper understanding of its inherent behaviors and patterns. This approach aims to detect not only known malware but also uncover new, previously unidentified threats. A comprehensive comparison with traditional static analysis methods shows that this approach excels in key performance metrics such as accuracy, precision, recall, and F_1 score. These results demonstrate the significant potential of combining L-moments derived from network analysis with ASTs in enhancing malware detection. While static analysis remains an essential tool in cybersecurity, the integration of L-moments and advanced network analysis offers a more effective and efficient response to the dynamic landscape of cyber threats. This study paves the way for future research, particularly in extending the use of L-moments and network analysis into additional areas.

1. Introduction

Malware represents a significant, and costly, modern challenge. As of 2023, ransomware attacks cost small businesses between one and two million dollars per incident (Langlois et al., 2023). There are no simple solutions to the problem. Signature and heuristic based methods are often insufficient to counteract sophisticated malware (Buczak and Guven, 2016). In particular, because signature based solutions require an initial detection followed by creation and dissemination of appropriate signatures, they are inherently reactive. There is a pressing need to explore novel methods to augment traditional approaches in detecting and mitigating emerging malware threats. One such promising avenue lies in analyzing source code structure, specifically PowerShell based malware, using Abstract Syntax Trees (ASTs) and advanced statistical methods such as L-moments.

Attackers have exploited PowerShell to develop sophisticated malware scripts. By representing PowerShell code as an AST, researchers can analyze the structural characteristics of the code, allowing for a deeper understanding of its behaviors and patterns. This approach provides a critical foundation for other sophisticated detection methods, based on machine learning and neural network techniques, to classify code as benign or malicious (Ding et al., 2018; Hu et al., 2019; Pascanu et al., 2015; Saxe and Berlin, 2015). However, these sophisticated approaches come with high computational costs and heavily depend on preprocessing quality.

To address these challenges, this research proposes using L-moments, a robust statistical method that has proven useful in various fields, but remains relatively unexplored in malware classification. L-moments offer a resilient approach to estimating distribution parameters and summarizing distributional shapes (Hosking, 1990). They can enhance the analysis of ASTs for malware detection. By combining ASTs with L-moments, this research aims to devise a proactive approach to correctly classifying previously unseen PowerShell malware that can be translated into a generalized approach for other malware types.

This research introduces a novel approach to malware detection, leveraging L-moments and the structural insights provided by ASTs.

* Corresponding author.
E-mail addresses: anthony.rose@afit.edu (A.J. Rose), christine.schubert_kabban@au.af.edu (C.M.S. Kabban), scott.graham@afit.edu (S.R. Graham), Wayne.Henry@afit.edu (W.C. Henry), Christopher.Rondeau@afit.edu (C.M. Rondeau).
URL: https://www.AFIT.edu (A.J. Rose).

https://doi.org/10.1016/j.cose.2024.104082
Received 19 May 2024; Received in revised form 23 August 2024; Accepted 26 August 2024
Available online 6 September 2024

By focusing on the inherent structural characteristics of PowerShell code, this method transcends the limitations of traditional detection systems, offering a more robust and versatile framework for identifying both known and unknown malware. This paper details the development and evaluation of the model, demonstrating its potential to outperform existing solutions, including Windows Defender.

The remainder of this paper is organized as follows. Section 2 provides a comprehensive background on L-moments and ASTs, laying the foundation for their application in malware detection. Section 3 outlines how L-moments are applied to ASTs through network analysis using centrality measures. Section 4 elaborates on data collection and model construction. Section 6 presents the model evaluation metrics, demonstrating the effectiveness of this approach through accuracy, precision, recall, and F_1 score analysis. Section 7 builds and validates the model with an existing signature based detection method. Finally, Section 8 concludes the paper with a summary of contributions and suggests future malware detection research.

2. Background

Signature and heuristic based analysis require malware to be identified first to create a detection. Though effective to some extent, these methods have become increasingly inadequate in detecting advanced forms of malware due to their inability to adapt swiftly to the continuously evolving threat landscape (Buczak and Guven, 2016). This motivates a shift towards other innovative malware detection techniques. Two such avenues are analyzing ASTs and applying statistical techniques such as L-moments.

In computer science, ASTs have been a fundamental tool since their introduction, primarily utilized in compiler design and programming language theory (Irons, 1961). Introduced as a notation to describe the syntax of programming languages, they have since found extensive usage in various domains (Irons, 1961). The representation of code as an AST allows for an in-depth analysis of its structural characteristics, providing insights into the behaviors and patterns embedded within the code. This understanding forms the basis of advanced detection systems to classify malware.

L-moments provide a resilient means of estimating distribution parameters and summarizing distributional shapes. The succeeding sections delve deeper into these key concepts, shedding light on their potential applications, existing research, and inherent challenges in the context of malware classification.

2.1. Previous works and limitations

A significant amount of prior research has explored the application of machine learning and neural networks to malicious code detection using ASTs. These studies leverage the structural information in the ASTs to classify malicious files (Fang et al., 2021; Li et al., 2019; Mimura and Tajiri, 2021; Rusak et al., 2018; Song et al., 2021). One notable work designed a deep learning model to classify Windows executable files (Saxe and Berlin, 2015). Their model successfully identified malicious binaries with high accuracy, demonstrating the potential of neural networks in this domain. However, they acknowledged the limitation of high computational costs associated with training deep learning models with only malicious data.

To complement existing research on the application of machine learning to malicious code detection via ASTs, a notable advancement comes from Tsai et al. in their work on PowerDP, which focuses on deobfuscating and profiling malicious PowerShell commands using multi-label classifiers (Tsai et al., 2023). While deep learning models exhibit potential in detecting malware through AST analysis, their success is significantly bound by the initial preprocessing steps, particularly deobfuscation.

Similarly, Pascanu proposed a technique to detect malicious JavaScript code by converting the code into ASTs and then applying Recurrent Neural Networks (RNNs) (Pascanu et al., 2015). They achieved promising results, although the computational expense of the approach was considerable due to the complex nature of RNNs. Ding et al. attempted to overcome this limitation by designing a lightweight Convolutional Neural Network (CNN) for malicious code detection (Ding et al., 2018). While their model was less computationally demanding than previous ones, they noted that the success of the method largely depended on the quality of the preprocessing steps, specifically the transformation of code into ASTs.

Graph Neural Networks (GNNs) were recently used to detect malware in Android applications by modeling the their behavior as Function Call Graphs (FCGs) (Lo et al., 2022). This approach showed a promising detection rate, but again, the results were highly reliant on preprocessing quality, and computational costs remained a significant concern. While existing methods utilize more complex graph-based representations and raw code to detect malware, these works demonstrate that such approaches can be effective (Hendler et al., 2018, 2020; Mimura and Tajiri, 2021). Given that ASTs are also graph representations of code, they hold similar potential for malware detection when analyzed with machine learning techniques. However, they also highlight a consistent challenge that these techniques are computationally expensive and performance is critically dependent on the preprocessing quality. Consequently, there is a pressing need to either enhance the effectiveness of preprocessing methods or consider eliminating preprocessing altogether, in order to reduce computational requirements and improve the overall robustness and efficiency of malware classification.

2.2. Abstract syntax trees

ASTs represent the syntactic structure of source code in a hierarchical tree-like format, with each node of the tree denoting a construct in the source code (Aho et al., 2006). The root of the tree represents the entire program, with leaf nodes corresponding to the terminal symbols like identifiers and constants and non-leaf nodes to the non-terminals like operators and keywords of the program. For example, a simple 'Hello World' script is illustrated in Fig. 1, demonstrating the hierarchical nature of ASTs by breaking the script into its syntactic elements.

One of the key features of ASTs is their ability to abstract away low-level syntactic details. For instance, ASTs do not contain explicit information about parentheses or certain details about how the code is formatted. This makes ASTs valuable for tasks where the logical and structural characteristics of the code are more important than its specific lexical or syntactic representation.

ASTs are fundamental in the implementation of compilers and interpreters. They serve as an intermediary form between the source code and the generated code, allowing the compiler or interpreter to analyze and transform the code more effectively. The application of ASTs has extended beyond compiler design into other areas, such as code refactoring, static code analysis, and malware detection. Various machine learning and data mining techniques can examine an AST to understand and predict code properties. In the context of malware detection, ASTs can provide insights into the behaviors and patterns embedded in the malicious code, forming a critical part of sophisticated detection methods.

2.3. PowerShell

PowerShell, a scripting language built on the .NET Framework, utilizes the framework's classes and libraries to automate administrative tasks (Microsoft, 2024c). A crucial feature of PowerShell is its use of the .NET Framework's Common Language Runtime (CLR) for executing commands. Commands in PowerShell are compiled into bytecode, which is then executed by the CLR. This execution process involves parsing PowerShell scripts into an AST, which represents the script's structure in a hierarchical form. The AST allows for the analysis and execution of scripts by breaking down and understanding the syntax

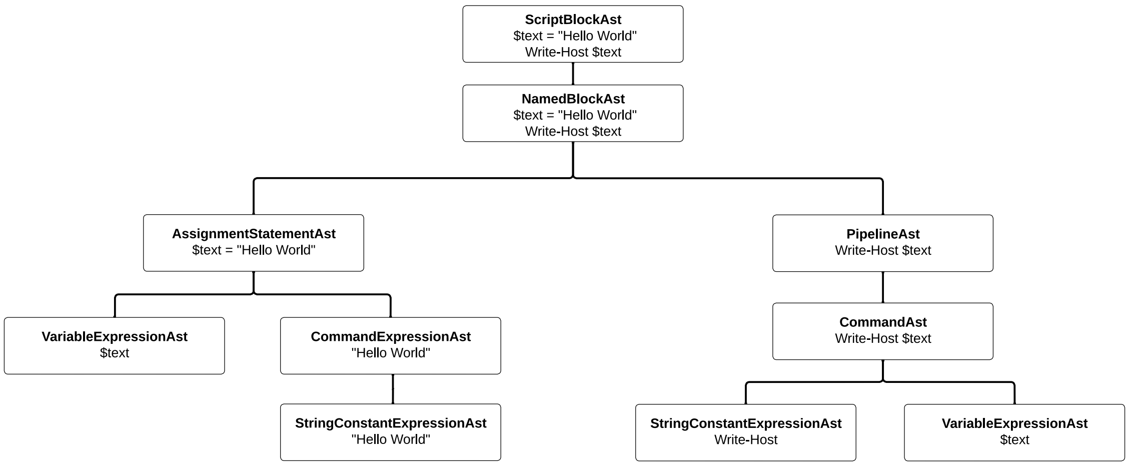


Fig. 1. Hierarchical representation of a PowerShell AST for a simple ‘Hello World’ script.

and semantics of the code. By leveraging the .NET Framework’s capabilities through the CLR, PowerShell efficiently processes and executes commands, making it a powerful tool for automation in .NET based environments.

2.3.1. Key elements of PowerShell ASTs

The PowerShell AST consists of several essential components that delineate the structure and behavior of PowerShell scripts. These elements include ScriptBlockAst, CommandAst, ExpressionAst, and ParameterAst. The ScriptBlockAst serves as the tree’s root node and contains the entire script or script block, encapsulating the logical flow of the script. CommandAst represents individual commands or cmdlets in the script, defining their structure and parameters. Cmdlets are specialized commands in PowerShell designed to perform a specific function, like managing system processes or manipulating data. ExpressionAst, on the other hand, is responsible for handling various expressions, such as arithmetic operations, variable assignments, and function calls. Finally, the ParameterAst deals with the parameters associated with commands and functions, defining their names, types, and default values.

2.3.2. ScriptBlockAsts

The ScriptBlockAst element is the cornerstone of PowerShell ASTs. It assumes the role of the root node within the AST hierarchy (Microsoft, 2023d). This encapsulation includes an array of other AST elements, such as CommandAst, ExpressionAst, and ParameterAst, collectively representing the script’s commands, expressions, parameters, and nested script blocks. The ScriptBlockAst acts as a container for dynamic script generation for runtime execution based on conditional logic or user input and establishes security boundaries to isolate potentially unsafe code. The hierarchical nature of ScriptBlockAst facilitates a structured representation of the PowerShell script, offering a high-level view of the script’s logical flow and structural organization.

2.3.3. CommandAsts

CommandAsts represent, among other things, individual commands or cmdlets within a script (Microsoft, 2023a). They are the primary means for performing actions, executing tasks, and interacting with system resources. CommandAsts capture the structure and semantics of a command, encompassing its name, arguments, and parameter bindings. In addition to built-in cmdlets, CommandAsts can also represent user-defined functions, external executables, or script blocks, allowing for a wide range of functionality and versatility in PowerShell scripts. CommandAsts give insight into the sequence of operations in a script and identify potential issues, such as incorrect parameter usage, nonexistent cmdlets, or unintended side effects. Moreover, CommandAsts facilitate script optimization and refactoring, enabling the modification and rearrangement of commands to enhance script readability, maintainability, and performance.

2.3.4. ExpressionAsts

ExpressionAsts represent various expressions within a script, including arithmetic operations, logical comparisons, variable assignments, and function calls (Microsoft, 2023b). As the building blocks of more complex constructs, ExpressionAsts help define the behavior and data manipulation in PowerShell scripts. They come in different types, such as constant expressions (e.g., literals and variables), binary expressions (e.g., addition or comparison), and invocation expressions (e.g., method or function calls), each with its unique characteristics and representation. ExpressionAsts provide a perspective on the data manipulation and processing in a script, enabling them to identify potential issues such as type mismatches, incorrect calculations, or unintended data transformations. Furthermore, ExpressionAsts play a significant role in script optimization as they can be refactored or simplified to enhance script readability, maintainability, and performance.

2.3.5. ParameterAsts

ParameterAsts represent the parameters associated with commands and functions within a script (Microsoft, 2023c). They define parameter names, types, default values, and attributes such as mandatory, position, or pipeline input specifications. ParameterAsts contribute to the modularity and reusability of PowerShell scripts by facilitating the passing of values and data between different script components, enabling customizable behavior and dynamic data manipulation. Analyzing ParameterAsts offers a view into the interaction between various script elements and validates the proper usage of parameters, ensuring that required parameters are provided, optional parameters have suitable default values, and data types are correctly handled. Furthermore, ParameterAsts support script optimization and refactoring by allowing the identification and modification of redundant or poorly designed parameters, ultimately enhancing the readability, maintainability, and performance of PowerShell scripts.

2.4. L-moments

L-moments are linear combinations of order statistics introduced by Hosking in 1990 as a new approach for summarizing distributional shapes (Hosking, 1990). This concept was proposed as an alternative to traditional statistical moments, which, while beneficial in various contexts, can be heavily influenced by outliers and extreme values in the dataset.

L-moments are fundamentally linear combinations of expectations of order statistics, where order statistics are the values of an ordered

[1] See Eq. (1) for an expression for the ith PWM, β_r.

Table 1
Comparative analysis of the first four L-moments and their corresponding traditional statistical moments[1].

L-Moment	Equation	Comparison to traditional moments
L-Mean	$\lambda_1 = \beta_0$	Equivalent to the arithmetic mean of the data.
L-Variance	$\lambda_2 = 2\beta_1 - \beta_0$	Analogous to traditional variance, but less affected by extreme values.
L-Skewness	$\lambda_3 = 6\beta_2 - 6\beta_1 + \beta_0$	Reflects the asymmetry of the data distribution.
L-Kurtosis	$\lambda_4 = 20\beta_3 - 30\beta_2 + 12\beta_1 - \beta_0$	Indicates the peakedness of the data distribution.
L-Skewness ratio	$\tau_3 = \frac{\lambda_3}{\lambda_2}$	Standardized L-Skewness.
L-Kurtosis ratio	$\tau_4 = \frac{\lambda_4}{\lambda_2}$	Standardized L-Kurtosis.

data set, sorted in ascending order (Hosking, 1990). The primary advantage of L-moments over traditional moments lies in their robustness to extreme values and their ability to provide a more reliable estimate of parameters in various distributional forms (Guttman, 1993). The first four L-moments, analogous to the first four traditional moments, are L-mean (location), L-variance (scale), L-skewness (asymmetry), and L-kurtosis (peakedness) (Hosking and Wallis, 1997). It should be noted, however, that with the exception of L-mean, the L-moments do not directly map onto the traditional moments due to their different computational methods.

Calculating L-moments is a crucial step in the analysis, and it begins with the estimation of the Probability-Weighted Moment (PWM) (Hosking, 1997). Let us consider a dataset $x_1, x_2, \ldots, x_n$, where x_i represents the ith smallest value in the set. The rth PWM, denoted as β_r, is a foundational component in the calculation of L-moments and can be estimated using Eq. (1).

$$\beta_r = \frac{1}{n} \sum_{i=1}^{n} \left(\frac{n-i}{n-1} \right)^r x_i \tag{1}$$

The PWMs provide a basis for computing L-moments, which are linear combinations of these PWMs. Therefore, the first L-moment, for instance, is simply the mean of the data, while higher-order L-moments reflect aspects of the data distribution such as variability, skewness, and kurtosis. L-moments are less influenced by extreme values compared to traditional moments, making them more reliable for statistical analysis, especially in the context of malware detection, where data distributions can be highly irregular (Hosking, 1990). To further elucidate, the kth L-moment, denoted as λ_k, can be defined in relation to PWMs. The equations for λ_1 through λ_4 can be found in Table 1.

In the context of L-moments, τ_3 and τ_4 are normalized forms of the third and fourth L-moments, respectively, as illustrated in Table 1. These normalized L-moments are particularly useful in statistical analysis as they provide standardized measures of skewness and kurtosis, which are less sensitive to outliers and extreme values compared to traditional moments (Vargo et al., 2010). Furthermore, both τ_3 and τ_4 offer valuable insights into the shape of data distributions as they can uniquely define a distribution.

This formulation underscores the relevance of PWMs in the computation of L-moments, illustrating how these statistical measures provide valuable insights into the structural characteristics of ASTs in malware detection. The concept of L-moments has been widely applied in various fields since its introduction. For example, in hydrology, L-moments have been used for flood frequency analysis due to their resilience to outliers and skewed distributions (Hosking and Wallis, 1997). Similarly, in finance, L-moments have been applied to model financial returns, which often exhibit non-normal behaviors (Fallahgoul et al., 2023). Despite the established use of L-moments in these fields, their application in cybersecurity, particularly in malware detection, remains relatively underexplored. This research seeks to leverage L-moments in this novel context.

3. Applying L-moments to abstract syntax trees for malware detection

The application of L-moments to ASTs for malware detection represents a novel approach to enhancing static code analysis techniques. The premise of this approach lies in the idea that the structure of the code, as represented by an AST, can carry significant information about the benign or malicious nature of the code. As such, if extracted and appropriately quantified, these structural characteristics can identify unique patterns or distributions that differentiate benign code from malicious code. In order to apply L-moments to ASTs, the AST must first be characterized as a network using mathematical graph theory.

3.1. Network analysis on ASTs

Graph theory provides powerful tools for representing and analyzing complex systems, particularly in the domain of computer science and software analysis. The network structure, in the form of nodes & edges, becomes an abstract representation of various elements such as data, relationships, dependencies, and hierarchies (Diestel, 2017). One of the most common forms of this graph representation is that of an AST (Aho et al., 2006). By applying the centrality measures outlined in Table 2 to the graphical representation of the ASTs, meaningful features indicative of the script's behavior can be extracted and used for malware detection.

For instance, the degree centrality of a node in an AST might reflect the complexity of a corresponding code construct, with high-degree nodes potentially signaling malicious obfuscation attempts. This notion aligns with the findings of Concas et al. who observed that software exhibits a power-law distribution in certain structural characteristics, suggesting that high-degree nodes could signal significant complexity or irregularity within the code structure (Concas et al., 2007). Similarly, the application of centrality measures in understanding software behavior is further supported by Kinable and Kostakis, who demonstrated the effectiveness of malware detection through the analysis of call graphs, a concept closely related to ASTs (Kinable and Kostakis, 2011). Their work emphasizes how nodes with high betweenness centrality correspond to critical operations or decision points and can be crucial in identifying potential malware.

The structure of the AST as a graph, with nodes representing code constructs and edges representing their syntactic relationships, allows the application of these network analysis measures. The analysis of the graph allows for informative malware detection features to be extracted (Chen and Deng, 2022). The network measures, including closeness, betweenness, degree, and other forms of centrality, can provide an abstract yet insightful representation of the code's structure and logic.

As part of our proposed approach, these network measures can be used in conjunction with L-moments to capture the distributional properties of the AST derived features. The L-moments of these measures, calculated across multiple ASTs, can provide a statistical summary of the code's structural features, which in turn can be used to differentiate between benign and malicious code. The following subsections define these network measures.

3.1.1. Degree centrality

Degree centrality is a fundamental concept in network analysis, providing a straightforward measure of a node's connectedness. In the context of ASTs, each node represents a construct in the source code, and the edges represent the syntactic relationship between the constructs. Therefore, the degree of centrality becomes particularly insightful. This research treats ASTs as undirected graphs. Consequently, a node's degree in an AST, denoted by $deg(v)$, quantifies the total number of direct syntactic connections a construct has with other constructs in the source code. Fig. 2 demonstrates that node A would have a high degree centrally for this graph with a value of 8.

Table 2
Descriptions of key centrality measures in network analysis.

Measure	Description
Degree Centrality	Degree centrality is perhaps the simplest and most intuitive measure of centrality. For a given node, its degree is the number of edges connected to it. In directed graphs, we can distinguish between in-degree (number of incoming edges) and out-degree (number of outgoing edges) (Freeman, 1978).
Closeness Centrality	Closeness centrality measures how close a node is to all other nodes in the network. Specifically, the closeness centrality of a node is the reciprocal of the sum of the shortest path distances from the node to all other nodes in the network (Sabidussi, 1966).
Betweenness Centrality	Betweenness centrality quantifies the extent to which a node acts as a bridge along the shortest paths between other nodes. Nodes with high betweenness centrality often have considerable influence over the network by controlling information flow (Freeman, 1977).

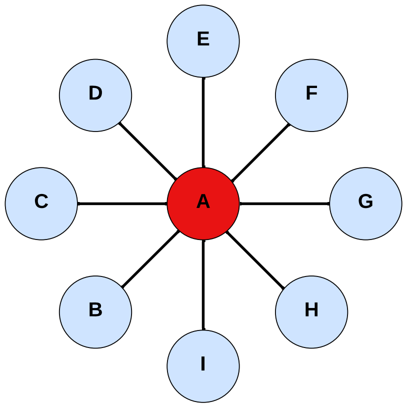


Fig. 2. Illustration of node *A* as the central hub in a network, emphasizing its degree centrality.

For our purposes, the degree centrality of a node, $C_D(v)$, is calculated using Eq. (2). This approach simplifies the analysis by focusing on the total number of connections each node has, regardless of their directionality. A high degree of centrality in an AST node may indicate a complex or significant code construct with numerous syntactic relationships. Such insights are invaluable for understanding the structural and behavioral aspects of code, especially in the context of malware classification.

$$C_D(v) = deg(v) \tag{2}$$

3.1.2. Closeness centrality

Closeness centrality is another critical measure in network analysis that highlights a node's proximity to other nodes in the network. Specifically, in an AST, closeness centrality can illustrate the relative distance of a code construct (node) from all other constructs in the source code. This measure is particularly insightful when considering the propagation of effects or influences within the source code, akin to information flowing within a network.

As demonstrated in Fig. 3, Node E exhibits the highest closeness centrality in the depicted AST. The closeness centrality, $C_C(v)$, of a node v is computed by taking the reciprocal of the sum of the shortest path distances from the node v to all other nodes t in the AST. The shortest path distance, $d(v,t)$, is the smallest number of edges that must be traversed to travel from node v to node t. The formula for calculating

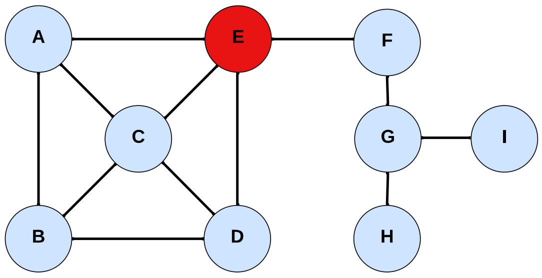


Fig. 3. Demonstrating closeness centrality with node *E* as the most proximal node in the network.

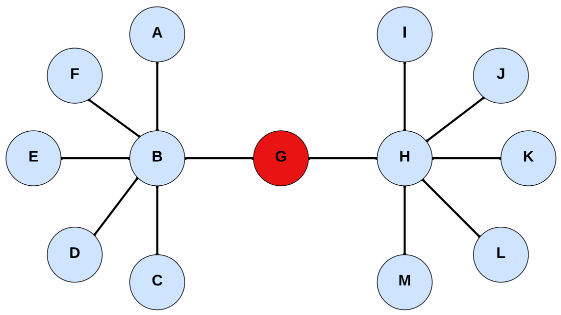


Fig. 4. Network diagram highlighting node *G* as a central broker illustrating betweenness centrality.

closeness centrality for a specific node is shown in Eq. (3) with node *E* having a closeness centrality of 0.071.

$$C_C(v) = \frac{1}{\sum_t d(v,t)} \tag{3}$$

Nodes with high closeness centrality, like node *E*, are positioned such that they can either influence or be influenced by many other nodes in fewer steps, owing to their central location. Identifying such nodes is key in analyzing the overall structure and behavior of source code, particularly for pinpointing vital constructs that could significantly affect the execution flow. Thus, closeness centrality can be leveraged in malware classification to identify potential focal points of malicious activities within the code structure.

3.1.3. Betweenness centrality

Betweenness centrality is a network analysis measure that evaluates a node's role as a bridge or gatekeeper within a network. In the context of ASTs, betweenness centrality can highlight those code constructs as nodes that serve as critical junctures in the control flow or data flow of the source code. Fig. 4 demonstrates that node *G* has high betweenness centrality to the graph, with node *G* having a betweenness centrality of 0.5.

The betweenness centrality, $B(v)$, of a node v is calculated as the sum of the fraction of all-pairs' shortest paths that pass through v. For any pair of nodes s and t, where s is not equal to v and t is not equal to v, σ_{st} is the total number of shortest paths from s to t, and $\sigma_{st}(v)$ is the number of those paths passing through v. Eq. (4) calculates betweenness centrality.

$$B(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{4}$$

Nodes with high betweenness centrality are often instrumental in controlling the flow within the source code, analogous to controlling the flow of information in a network. These nodes may be pivotal in orchestrating malicious activities within a piece of malware, dictating how different parts of the code interact with each other.

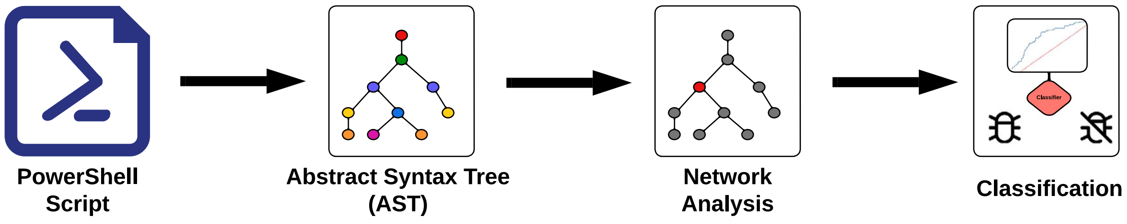


Fig. 5. Process flowchart for malicious PowerShell script classification through AST and network analysis.

4. Model construction methodology

The proposed methodology for classifying PowerShell scripts as benign or malicious begins with generating an AST that deconstructs the script into a structured representation of its code, as illustrated in Fig. 5. Following AST generation, key features are extracted, including node attributes, tree depth, and network measures. These features are then summarized using L-moments, which capture the statistical properties of the feature's distribution. These L-moments are subsequently fed into a logistic regression model to classify the script based on its structural properties.

The methodology evaluates the approach's computational performance, specifically analyzing the impact of factors such as node count and file size impact metrics like processing time. This process integrates structural analysis, statistical evaluation using L-moment calculations, and targeted performance optimization.

4.1. AST generation

The classification process begins by transforming a given piece of source code into an AST. This transformation uses tools that parse the source code according to the programming language's grammar. The resulting AST is a tree-like structure that abstracts away syntactic details, preserving the code's structural and logical elements.

System.Management.Automation.Language.Parser provides a built-in ParseFile function for PowerShell scripts, capable of converting raw script text into an AST by following the syntax rules of the language (Microsoft, 2024b). The System.Management.Automation.Language.Ast class defines the structure of all PowerShell AST nodes, serving as the abstract base class for all nodes (Microsoft, 2024a). This class provides the foundation for the hierarchical representation of the script, where each node represents a syntactic element of the code.

After the AST is generated, it is serialized into an Extensible Markup Language (XML) format. The serialization process involves recursively mapping each child node to its parent, as shown in the simplified version of the PowerShell code in Listing 1. The PopulateNode and AddChildNode functions handle this recursive mapping by iterating through the properties of each node, adding attributes to the XML structure. Once serialized, the entire AST reflects the syntactic relationships and structural hierarchy of the original script.

4.2. Feature extraction

After the AST is generated and serialized into an XML format, the next step involves extracting relevant features that capture the structural and logical characteristics of the code. The XML representation of the ASTs are ingested into the Python NetworkX library, where they are converted into graph structures. In this representation, each node in the AST corresponds to a node in the graph, and the edges between nodes represent the syntactic relationships within the code. This graph-based representation allows for the calculation of various network measures.

The features extracted from the ASTs include node attributes such as the number of nodes, the types of nodes (e.g., function calls, variable declarations), and the distribution of these node types throughout the

Listing 1 Simplified PowerShell script to convert a PowerShell file (PS1) to XML AST.

```
function PopulateNode($xmlWriter, $object) {
    foreach ($child in $object.PSObject.Properties) {
        if ($child.Name -eq 'Parent') {
            continue
        }

        $childObject = $child.Value

        if ($childObject -is [System.Management.Automa⌋
         ↪ tion.Language.Ast])
         ↪ {
            AddChildNode $xmlWriter $childObject
            continue
        }
        ...
    }
}

function AddChildNode($xmlWriter, $child) {
    $global:n_nodes += 1
    $xmlWriter.WriteStartElement($child.GetType().Name)

    foreach ($property in $child.PSObject.Properties) {
        if ($property.Name -in 'Name', 'TypeName',
         ↪ 'Operator') {
            $xmlWriter.WriteAttributeString($property.⌋
             ↪ Name,
             ↪ [xml]::Escape($property.Value))
        }
        ...
    }
    PopulateNode $xmlWriter $child
    $xmlWriter.WriteEndElement()
}

function Convert-PowershellScriptToXmlAst {
    $AST = [System.Management.Automation.Language.Pars⌋
     ↪ er]::ParseFile($InputScript, [ref]$null,
     ↪ [ref]$null)

    $xmlsettings = New-Object
     ↪ System.Xml.XmlWriterSettings
    $XmlWriter =
     ↪ [System.XML.XmlWriter]::Create($OutputXmlFile,
     ↪ $xmlsettings)

    AddChildNode $xmlWriter $AST
    ...
}
```

AST. Additionally, the depth of the tree is computed, which represents the maximum level of nested code within the script. The tree depth provides insights into the complexity of the code and potential obfuscation techniques used by malicious actors. The specific set of features extracted depends on the programming language and the characteristics of the malware being analyzed. For instance, in PowerShell scripts, nodes related to command execution or file operations are of particular interest.

Beyond basic node attributes and tree depth, more advanced network measures are calculated. These include degree centrality, betweenness centrality, and closeness centrality from Section 3.1, which provide insights into the significance and influence of specific nodes within the AST structure. For example, nodes with high betweenness centrality may represent critical decision points in the code, while nodes with high degree centrality may correspond to frequently used variables or functions. These network measures offer a deeper understanding of the code's structure.

4.3. L-moment calculation

Once the pertinent features are extracted from the ASTs, the next step is to calculate the associated L-moments for these features. The L-moments (L-mean, L-variance, L-skewness, L-kurtosis, and their ratios) are calculated using the lmoments3 library, which uses the equations from Section 4.2. The network measures obtained from the feature extraction phase (degree centrality, betweenness centrality, and closeness centrality) are treated as distributions for which the L-moments are computed. A detailed analysis of the L-moment calculations for the PowerShell ASTs is presented in Section 5.1.

As an example, L-moments calculated from PowerShell ASTs using degree centrality reveal how evenly distributed commands and variables are throughout the script, while L-moments for betweenness centrality indicate the presence of critical decision points that are central to the script's functionality. The use of L-moments in this context allows for a more detailed analysis of the AST features, capturing both the central tendencies and variability in the data. This is particularly useful when dealing with obfuscated or irregular code, as L-moments provide a way to summarize the data.

4.4. Classification

The final step in the process is classification, where the calculated L-moments serve as input to a model that is trained to distinguish between benign and malicious scripts. For this research, a logistic regression model is used due to its simplicity, interpretability, and effectiveness in binary classification tasks. Logistic regression is particularly well-suited for scenarios where the relationship between features and the target variable (benign vs. malicious) can be approximated as a linear combination of the input features. The model is trained on a dataset of labeled PowerShell scripts, where the L-moments of their AST features serve as the input. The results and analysis of the classification process are further discussed in Section 7.

5. Data collection and performance metrics

The dataset comprises 7014 PowerShell scripts, collected and categorized into non-malicious and malicious scripts. The resulting set has 5085 malicious and 1929 non-malicious scripts. Each script was transformed into an AST and analyzed to extract structural features, which were then utilized to compute L-moments for classification. The malicious scripts were obtained from publicly available datasets on GitHub and malware samples obtained from Malware Bazaar (Malware Bazaar, 2024; Security, 2024; Gill, 2024; Marra, 2024; Mittal, 2024; Nettitude, 2024; Rapid7, 2024; Palo Alto - Unit42, 2024). In contrast, the non-malicious scripts, representing a broad spectrum of common PowerShell tasks (Albert, 2024; Andrews, 2024; Bekker et al., 2024; Johnson, 2024; MrPowerScripts, 2024; Splunk, 2024; PowerShell Team, 2024; VMware, 2024), provide a comprehensive baseline for normal script behavior.

Table 3
Distribution of non-malicious script types with count and proportion.

Script type	Count	Proportion
Active directory	264	0.14
Administration	389	0.20
Confluence	258	0.13
Files	55	0.03
Installers	454	0.24
Office	38	0.02
Scanner	66	0.03
Splunk	220	0.11
VMware	185	0.10
Total	1,929	1.00

Table 4
Distribution of malicious script types with count and proportion.

Script type	Count	Proportion
Command & Control	2029	0.40
Code execution	74	0.01
Collection	146	0.03
Credentials	92	0.02
Downloader	1,324	0.26
Exfiltration	48	0.01
Exploitation	54	0.01
Lateral movement	153	0.03
Management	342	0.07
Persistence	73	0.01
Privilege escalation	312	0.06
Situational awareness	309	0.06
Troll	129	0.03
Total	5,085	1.00

A crucial aspect of this research is the deliberate approach taken in training and testing the model. The primary goal of evaluating this technique against known threats is to estimate its effectiveness in detecting unknown threats. To achieve this, the model was trained on a subset of the dataset, including both malicious and non-malicious scripts, with stratified sampling used to preserve the original data distribution. This ensures that the test set, comprising 20% of the total data, remained entirely unknown to the model during the training phase. This separation mimics a real-world scenario where the tool encounters threats it has not been previously exposed to, effectively treating known threats as if they were unknown.

Furthermore, this methodology allows for comparative analysis against other tools, such as Windows Defender, that specialize in detecting known threats. By assessing the technique's performance under these conditions, valuable insights are gained into its relative effectiveness. The comparison is designed to determine whether this approach, which leverages the structural nuances captured by ASTs and L-moments, offers a significant advantage over existing methods in identifying malware, particularly in instances where the threats are not known. Tables 3 & 4 provide a detailed distribution of the scripts used in the study, highlighting the comprehensive nature of the dataset and the rigorous approach taken in its analysis.

These tables illustrate the breakdown of script types and their respective counts and proportions, providing a clear view of the distribution of non-malicious and malicious scripts within the dataset. This data is pivotal for the subsequent phases of feature extraction, L-moment calculation, and classification. By understanding the typical distributions and characteristics of these scripts, the model can be better tuned to detect subtle nuances that distinguish benign from malicious scripts.

5.1. L-moment calculations for PowerShell ASTs

Notable statistical distinctions in the centrality moments between obfuscated and non-obfuscated scripts emerged when analyzing the L-moments. The comparison of script metrics, including malicious and

Table 5
Comparison of script metrics including malicious and obfuscation status and the first four *degree* centrality moments.

Name	Nodes	Malicious	Obfuscated	λ_1	λ_2	τ_3	τ_4
Invoke-Assembly	247	✓		0.0081	0.0022	0.2303	0.0248
Invoke-Assembly	774	✓	✓	0.0026	0.0008	0.3413	0.1265
Invoke-DllInjection	1,743	✓		0.0011	0.0003	0.2858	0.0706
Invoke-DllInjection	4,286	✓	✓	0.0005	0.0001	0.3281	0.1189
Get-OfficeInstalledVersion	98			0.0204	0.0055	0.1854	−0.0345
Get-OfficeInstalledVersion	295		✓	0.0068	0.0019	0.2508	0.0480
Export-ScheduledTask	204			0.0098	0.0028	0.2547	0.0169
Export-ScheduledTask	415		✓	0.0048	0.0014	0.2636	0.0536

Table 6
Comparison of script metrics including malicious and obfuscation status and the first four *closeness* centrality moments.

Name	Nodes	Malicious	Obfuscated	λ_1	λ_2	τ_3	τ_4
Invoke-Assembly	247	✓		0.0756	0.0088	0.0834	0.0864
Invoke-Assembly	774	✓	✓	0.0420	0.0050	0.1627	0.0764
Invoke-DllInjection	1,743	✓		0.0676	0.0096	0.0342	0.1354
Invoke-DllInjection	4,286	✓	✓	0.0469	0.0062	0.1451	0.1194
Get-OfficeInstalledVersion	98			0.0895	0.0109	0.0536	0.0437
Get-OfficeInstalledVersion	295		✓	0.0511	0.0072	0.1260	0.0788
Export-ScheduledTask	204			0.0763	0.0087	0.1254	0.1086
Export-ScheduledTask	415		✓	0.0493	0.0065	0.0918	0.0691

Table 7
Comparison of script metrics including malicious and obfuscation status and the first four *betweenness* centrality moments.

Name	Nodes	Malicious	Obfuscated	λ_1	λ_2	τ_3	τ_4
Invoke-Assembly	247	✓		0.0523	0.0386	0.5999	0.3874
Invoke-Assembly	774	✓	✓	0.0309	0.0234	0.6168	0.3870
Invoke-DllInjection	1,743	✓		0.0086	0.0065	0.6310	0.4248
Invoke-DllInjection	4,286	✓	✓	0.0050	0.0037	0.6071	0.3961
Get-OfficeInstalledVersion	98			0.1113	0.0763	0.4680	0.1735
Get-OfficeInstalledVersion	295		✓	0.0675	0.0460	0.4628	0.1820
Export-ScheduledTask	204			0.0625	0.0474	0.5987	0.3309
Export-ScheduledTask	415		✓	0.0494	0.0362	0.5575	0.2890

obfuscation status across the first four degree centrality moments, closeness centrality moments, and betweenness centrality moments, reveals insights into the nature of these scripts. A sample of the data is shown in Tables 5, 6, & 7.

A critical observation is the significant difference in the L-moments between obfuscated and non-obfuscated scripts. While obfuscation does not inherently indicate malicious intent, it is rare for legitimate PowerShell scripts to employ obfuscation unless for specific, justified reasons. This distinction becomes particularly pronounced when examining the τ_3 value of the degree centrality. The data suggests a statistically significant difference when comparing τ_3 between malicious and non-malicious scripts, indicating the potential of τ_3 as a discriminating feature in identifying malicious activity within PowerShell scripts.

Moreover, the data shows that combining features rather than a single metric is necessary for a more accurate and robust model for classifying malicious scripts. Analyzing the degree, closeness, and betweenness centrality moments provides a multifaceted view of the scripts' structural properties. The significant differences observed in the L-moments, particularly in the context of obfuscation, offer promising directions for further research and model development.

Table 5 compares the τ_3 values for degree centrality across different scripts with varying malicious and obfuscation statuses. For example, the Invoke-Assembly script shows an increase in τ_3 from 0.2303 for the non-obfuscated version to 0.3413 for the obfuscated version, indicating increased structural complexity due to obfuscation. This trend is consistent for other scripts in the table, reinforcing the hypothesis that obfuscated scripts exhibit a higher degree of structural irregularities, potentially making them more difficult to detect using simplistic models.

5.2. Data calculation performance metrics

The performance evaluation of malware detection systems is critical to understanding their efficiency and scalability. This section analyzes the impact of various factors, such as node count and file sizes, on the performance of different analysis techniques used in detecting malicious and non-malicious PowerShell scripts. As discussed in Section 5, a total of 7014 samples were analyzed to assess these performance metrics. These factors are used to identify the computational challenges and resource requirements associated with processing complex scripts. This work processed the calculations on a Ryzen 9 5950x with 128 GB of RAM.

5.2.1. Time impact

This section presents the summary statistics for execution times during the three data generation phases for non-malicious and malicious scripts. The phases include generating the AST, graph generation, and network analysis. Table 8 and 9 summarize the execution times for non-malicious and malicious scripts, respectively.

The first step in the process requires evaluating a PowerShell script, converting it into an AST, and saving it as an XML. Fig. 6(a) shows that the time variation can be accounted for by differences in the processor's workload and performance during analysis due to the small processing time. The average impact on time for non-malicious scripts is 6.16 s and 7.23 s for malicious scripts.

Fig. 6(b) presents the time taken for graph analysis. The graph time includes ingesting the XML file from the previous step, converting it into an AST, and representing it as a graph. The time increases significantly with the number of nodes, especially for malicious scripts. This indicates that the graph structures of malicious scripts are more complex, possibly involving multiple relationships and interactions that need to be processed, leading to higher computational costs. The average impact on time for non-malicious scripts is 0.14 s and 0.45 s for malicious scripts.

Finally, Fig. 6(c) highlights the time required to perform network analysis on each node and compute centrality measures. This step

Table 8
Summary statistics for execution times in seconds for non-malicious scripts.

Statistic	Generate AST	Graph generation	Network analysis	Total time
mean	6.16	0.14	7.99	14.29
std	2.96	0.77	46.57	48.47
min	1.87	0.00	0.00	1.88
25%	4.11	0.00	0.01	4.57
50%	5.28	0.01	0.17	6.02
75%	7.46	0.03	1.51	8.34
max	22.84	14.45	734.77	760.43

Table 9
Summary statistics for execution times in seconds for malicious scripts.

Statistic	Generate AST	Graph generation	Network analysis	Total time
mean	7.23	0.45	32.57	40.26
std	4.52	3.70	327.63	331.98
min	1.40	0.00	0.00	1.43
25%	3.66	0.00	0.02	3.84
50%	6.62	0.02	0.95	9.98
75%	9.72	0.13	6.60	16.56
max	52.14	102.97	8, 603.78	8, 689.13

Table 10
Summary statistics for file sizes in bytes for non-malicious scripts.

Statistic	PS1 size	XML size
mean	8,084.48	58,023.51
std	16,037.51	298,130.47
min	56.00	178.00
25%	1,458.00	1,675.25
50%	3,338.00	9,795.00
75%	8,233.50	37,546.25
max	231,262.00	5,427,008.00

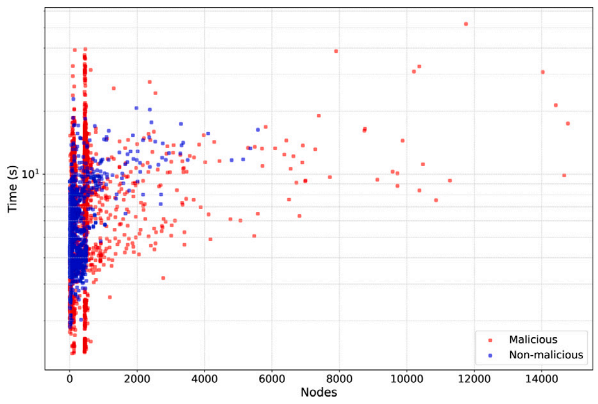
Table 11
Summary statistics for file sizes in bytes for malicious scripts.

Statistic	PS1 size	XML size
mean	23,549.76	56,323.85
std	260,950.46	284,411.29
min	8.00	178.00
25%	154.00	2,917.00
50%	2,231.00	22,101.00
75%	2,331.00	37,988.00
max	8,286,690.00	9,315,214.00

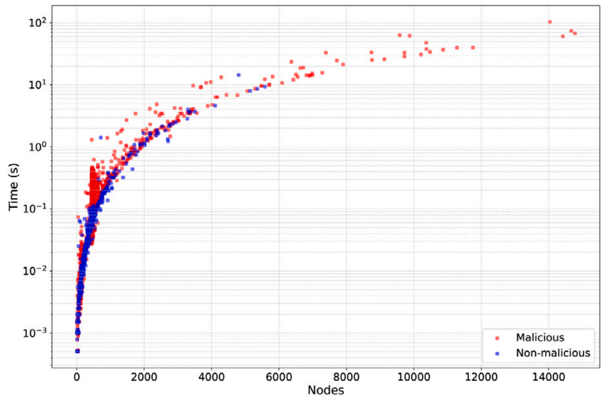
also includes the computation of L-moments, which, although combined with this step, contribute insignificantly with less than 0.1 s of computational time. The centrality time, shown on a logarithmic scale, indicates a steep increase, particularly for malicious scripts, as node count grows. This steep rise can be attributed to the intricate network structures present in malicious scripts, which require extensive calculations to determine node centrality. The average impact on time for non-malicious scripts is 7.99 s and 32.57 s for malicious scripts. The total time impact for using network analysis and L-moments for malware detection takes, on average, 14.28 s for a non-malicious script and 40.25 s for a malicious script. This method does not use optimization that Graphics Processing Units (GPUs) would bring and would significantly decrease the computational time.

5.2.2. *File size impact*

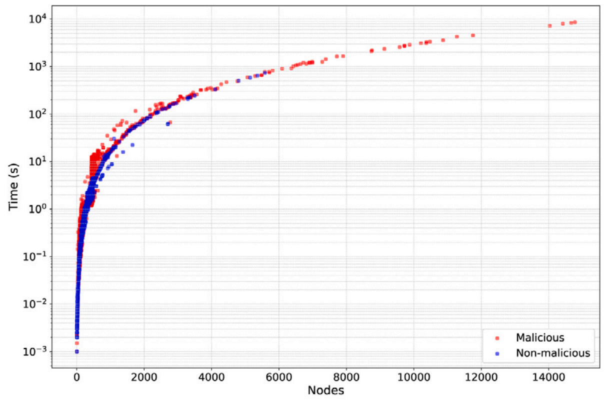
In addition to node count, the size of the files also significantly impacts the performance of analysis tasks, as seen in the summary statistics in Tables 10 and 11. Fig. 7(a) illustrates how the number of nodes affects file size. Malicious scripts generally have larger file sizes than non-malicious scripts, which can be attributed to the additional layers of obfuscation and complexity. As the file size increases, the time required for processing also rises, indicating a direct correlation between file size and computational demand.



(a) Impact on the number of nodes to time to extract AST from a PowerShell script.



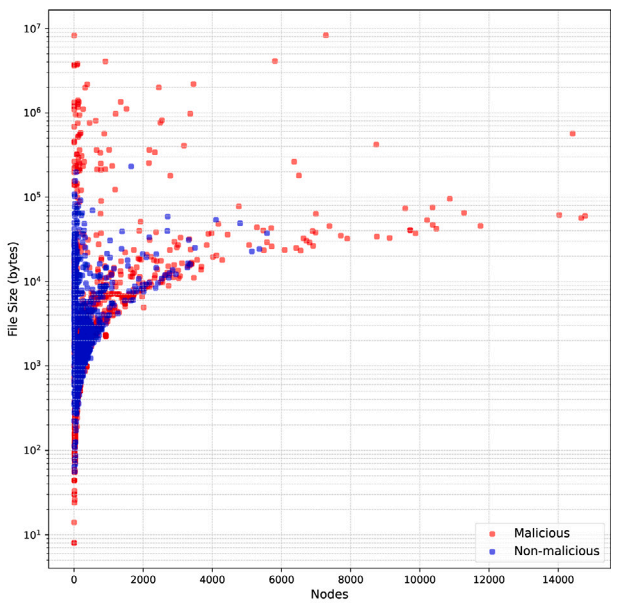
(b) Impact on the number of nodes to time to convert AST in XML format to a graph.



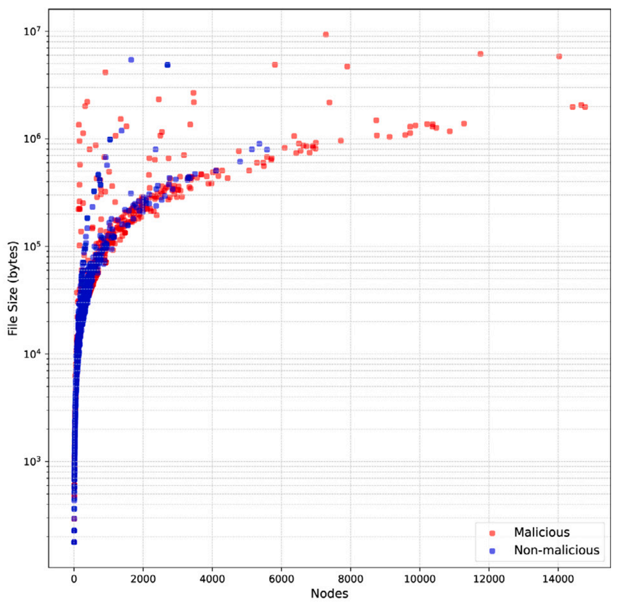
(c) Impact on the number of nodes to time to calculate network measures and moments from graph.

Fig. 6. Comparison of execution times for various phases of the classification methodology.

Similarly, Fig. 7(b) highlights the effect of nodes on the generated XML file size. XML file size becomes prominent in malicious scripts due to the inclusion of obfuscation and embedded resources. The analysis shows that the XML file size increases significantly as the number of nodes grows, particularly for malicious scripts.



(a) Impact on the number of nodes contained in the original PowerShell file compared to its size in bytes.



(b) Impact on the number of nodes to the size of the generated XML file in bytes.

Fig. 7. Comparison of original and generated file sizes and the number of nodes.

6. Model evaluation metrics

This section explores the key metrics used to evaluate the performance of classification models, providing insights into their accuracy and reliability, especially in the context of malware classification. Let T be the true script label denoted by 1 if malicious and 0 if non-malicious. Similarly, let Z be the predicted label. Assume there are n_1 malicious scripts, n_2 non-malicious scripts, and for a particular classification model, m_1 are classified as malicious, and m_2 are classified as non-malicious. Let N represent the whole sample size of the scripts $(n_1 + n_2 = N)$.

6.1. Accuracy

Accuracy is an essential metric for evaluating the overall effectiveness of a classification model. It is defined as the ratio of correctly predicted observations to the total number of observations, accounting for both True Positives (TPs) (truly malicious) and True Negatives (TNs) (truly non-malicious), as shown in Eq. (5) (Ghamrawi and McCallum, 2005; Zhu et al., 2005). While it provides a quick assessment of model performance, accuracy can be misleading in imbalanced datasets where it may not reflect the model's performance on the minority class.

$$\text{Accuracy} = \frac{1}{N}\sum_{i=1}^{N} I(Z_i = T_i) \tag{5}$$

where I is an indicator function that returns 1 for an exact match between T_i and Z_i, and 0 otherwise. This formulation of accuracy demands a complete match across all labels for a script to be considered correct, hence providing a stringent measure of a model's predictive power.

6.2. Precision

Precision, or the positive predictive value, is a metric for evaluating positive instances. It calculates the ratio of true positive predictions to the total positive predictions (including False Positives (FPs)), which is crucial in applications where FPs have significant consequences, such as medical diagnostics and spam detection.

$$\text{Precision} = \frac{1}{m_1}\sum_{i=1}^{m_1} I(Z_i = T_i | Z_i = 1) \tag{6}$$

Precision is particularly valuable in binary scenarios to gauge the accuracy of label predictions (Godbole and Sarawagi, 2004). Combined with recall, it offers a comprehensive view of model performance, especially in imbalanced datasets.

6.3. Recall

Recall, also known as sensitivity, is a metric used in binary classification to assess the model's ability to correctly identify actual positive instances. It is calculated as the proportion of correctly predicted positive labels, as shown in Eq. (7) (Maimon and Rokach, 2010).

$$\text{Recall} = \frac{1}{n_1}\sum_{i=1}^{n_1} I(Z_i = T_i | T_i = 1) \tag{7}$$

The recall value is especially important in applications where missing a TP can have serious implications, such as in medical diagnostics or security systems. For example, in medical testing, a high recall rate is crucial to ensure that as many actual cases of a disease as possible are detected. In security systems, recall helps to ensure that threats are not missed.

6.4. Specificity

Specificity, in a binary classification context, emphasizes the model's ability to correctly predict negative instances for each label. The specificity is calculated as the proportion of TN predictions to the actual negative instances, as expressed in Eq. (8).

$$\text{Specificity} = \frac{1}{n_2}\sum_{i=1}^{n_2} I(Z_i = T_i | T_i = 0) \tag{8}$$

This measure is vital for applications where the cost of FPs is significant. The methodology for computing binary specificity is informed by the work of Godbole and Sarawagi, who discussed the generalization of traditional metrics for the multi-label setting (Godbole and Sarawagi, 2004).

Table 12
Performance of defender on training and testing data as represented by confusion matrices.

Data type	Predicted		Actual	Accuracy	Precision	Recall	F1 score
	Non-malicious	Malicious					
Training data	1,515	28	Non-malicious	0.41	0.97	0.19	0.32
	3,289	779	Malicious				
Testing data	380	6	Non-malicious	0.42	0.97	0.20	0.33
	813	204	Malicious				

6.5. F_1 Score

The F_1 Score is a commonly used statistical measure in binary classification that combines precision and recall into a single metric. It is the harmonic mean of the precision and recall scores, giving equal weight to both FPs and False Negatives (FNs), which is particularly important in imbalanced datasets. The formula for calculating the F_1 Score in the context of binary classification is provided in Eq. (9) (Han et al., 2011; Maimon and Rokach, 2010).

$$F_1 = \frac{2 \times precision \times recall}{precision \times recall} \quad (9)$$

For example, in malware detection, where missing a positive instance (malware) could be more detrimental than incorrectly identifying a negative instance (benign software) as positive, F_1 helps to assess the model's effectiveness in identifying true malware cases while minimizing false alarms. It reflects the model's ability to balance the trade-off between capturing all positive instances and maintaining the accuracy of these predictions.

6.6. Windows defender's detection performance

The performance of Microsoft Defender in detecting malware provides a valuable case study for understanding key metrics like precision, recall, and the F_1 in threat detection. Defender's predictive behavior showcases a notable pattern. It demonstrates a higher proficiency in correctly identifying non-malicious instances, indicated by a significant number of TNs in both training and testing datasets. However, it notably underperforms in accurately classifying malicious instances, as reflected in the substantial count of FNs as seen in Table 12. This suggests a tendency towards conservative predictions, potentially at the cost of overlooking actual threats (Pogonin and Korkin, 2022).

The evaluation metrics, calculated from the testing data in Table 12, provide a more detailed understanding of Defender's performance. The accuracy of the Defender model is 0.42, which, while moderate, does not fully capture the model's predictive behavior. Notably, the precision is high at 0.97, suggesting that Defender classifies a script to be malicious relatively well. However, the recall is significantly lower at 0.20, indicating that Defender only correctly identifies a small number of malicious scripts. This low recall is a result of a high number of FNs. Furthermore, F_1 stands at 0.33, reflecting the challenges in balancing these two metrics.

These insights highlight that while Defender is quite effective in recognizing non-malicious scripts, its limitations in identifying true malicious instances are pronounced. This is particularly critical in balanced classification tasks where minimizing FNs is crucial. In the context of malware detection, missing a threat can have significant consequences, thereby emphasizing the need for models that are not only accurate but also have a balanced precision and recall.

7. Classification model

This research employs a logistic regression model to classify scripts as malicious or non-malicious. The logistic regression model is a probabilistic statistical classification model used to predict the probability of a binary response based on one or more predictor variables. The model is represented as shown in Eq. (10), where y is the probability that a given observation belongs to a particular class (in this case, malicious), e is the base of the natural logarithm, β_0 is the intercept, and $\beta_1, \beta_2, \ldots, \beta_n$ are the coefficients of the predictor variables $x_1, x_2, \ldots, x_n$, respectively. These coefficients represent the weights assigned to the features, which in our model include L-moments derived from network measures. The model was trained and validated on a dataset of ASTs as described in Section 5 to effectively identify known and unknown malware threats.

$$y = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n)}} \quad (10)$$

One of the significant challenges encountered during the training process was the imbalance in the dataset, with a larger proportion of malicious scripts compared to non-malicious ones. This imbalance can lead to biased model predictions, where the classifier may favor the majority class and overlook the minority class (Spelmen and Porkodi, 2018). Following the author's suggestion, Synthetic Minority Oversampling Technique (SMOTE) was applied to the training data. SMOTE generates synthetic examples of the minority class, non-malicious in this case, thereby balancing the class distribution (Chawla et al., 2002). It is important to note that the training data for Defender presented in Table 12 cannot be compared directly to the training data in Tables 13 & 14 due to the synthetic data applied during model training.

7.1. Hyperparameter tuning

This section is included to outline the hyperparameter tuning process. While it is not strictly necessary to aid the process, it allows future researchers to replicate and improve upon the research.

Hyperparameter tuning was applied to optimize the logistic regression model for detecting malicious and non-malicious PowerShell scripts. A grid search with cross-validation was used to explore a range of hyperparameters to identify the best combination for model performance (Bergstra and Bengio, 2012).

A standardized scalar was applied to the features by removing the mean and scaling to unit variance. This step is essential for models like logistic regression, which assumes that the input features are on the same scale. Standardizing the features helps faster convergence during model training and improves the model's performance. This method is linear, reversible, fast, and highly scalable, but it is sensitive to outliers and more suitable for normally distributed data (Ferreira et al., 2019). The standardized scalar, $\hat{X}_i$, can be calculated by taking each observation, X_i, from a sample with mean $\bar{X}$ and standard deviation σ, seen in Eq. (11).

$$\hat{X}_i = \frac{X_i - \bar{X}}{\sigma} \quad (11)$$

The regularization parameter, commonly referred to as C, controls the trade-off between achieving a low training error and a low testing error (Ng, 2004). A lower C value indicates stronger regularization, preventing overfitting by penalizing larger coefficients. Conversely, a higher C value means less regularization, aiming to fit the training data more closely.

During the hyperparameter tuning process, we used a grid search with cross-validation to identify the model's optimal C value. The C values of 0.01, 0.1, 1, 10, and 100 were tested. Setting the regularization parameter to 10 allowed the model to balance bias and variance, reducing the risk of overfitting while maintaining good generalization

Table 13
Performance metrics of logistic regression model using Youden's index, represented by a confusion matrix.

Data type	Predicted		Actual	Accuracy	Precision	Recall	F1 score
	Non-malicious	Malicious					
Training data	3,813	255	Non-malicious	0.81	0.92	0.68	0.78
	1,288	2,780	Malicious				
Testing data	359	27	Non-malicious	0.74	0.96	0.67	0.79
	337	680	Malicious				

Table 14
Evaluation of test data predictions using confusion matrix at optimal threshold set by custom cost function.

Data type	Predicted		Actual	Accuracy	Precision	Recall	F1 score
	Non-malicious	Malicious					
Testing data	380	6	Non-malicious	0.53	0.98	0.36	0.53
	647	370	Malicious				

performance. Additionally, we set the maximum iterations to 1000 to ensure that the optimization algorithm had sufficient iterations to converge.

The L1 penalty, also known as Lasso regularization, was applied to promote sparsity in the model coefficients. This means that by penalizing the absolute values of the coefficients, the L1 penalty drives many of them to zero, effectively selecting a subset of the most relevant features (Ng, 2004). This is particularly beneficial for feature selection and interpretability, as it simplifies the model by including only the most impactful features.

The Cholesky decomposition, used as part of the Newton-Cholesky solver, is an algorithm for solving linear equations, particularly suitable for positive definite matrices. Cholesky decomposition works by decomposing a matrix into a lower triangular matrix and its transpose. This method reduces computation time and improves numerical stability compared to other matrix inversion methods (Burian et al., 2003). In logistic regression, the Newton-Cholesky solver leverages this decomposition to enhance convergence rates and stability.

7.2. Youden's index

This model underwent further refinement with the application of Youden's index, also known as Youden's J statistic. Youden's index is used to ascertain the optimal balance between sensitivity and specificity in a binary classifier (Youden, 1950). It achieves this by maximizing the difference between the True Positive Rate (TPR) and the False Positive Rate (FPR), where Youden's index is defined as the sum of sensitivity and specificity minus one and is represented in Eq. (12).

$$J = \frac{1}{n_1}\sum_{i=1}^{n_1} I(Z_i = T_i|T_i = 1) + \frac{1}{n_2}\sum_{i=1}^{n_2} I(Z_i = T_i|T_i = 0) - 1 \qquad (12)$$

The model included an array of features, specifically the first four L-moments and L-moment ratios (λ_1, λ_2, τ_3, τ_4) for measures of degree centrality, betweenness centrality, and closeness centrality. Additionally, the fifth L-moment ratio (τ_5) was initially calculated, but ultimately removed from the final model. This decision was made based on the assessment that τ_5 did not significantly contribute to the model. Alongside these L-moment measures, the number of nodes in each network was also included as a feature, providing a foundational structural characteristic of the network for the model's analysis.

Applying Youden's index to the logistic regression model resulted in an optimized decision threshold of 0.67. This adjustment significantly improved the evaluation metrics, leading to an accuracy of 0.74, precision of 0.96, recall of 0.67, and an F_1 of 0.79, shown in Table 13. These improvements, particularly in recall and precision, demonstrate that this method not only matches the reliability of the signature based detection method in Defender, but also surpasses it in distinguishing between malicious and non-malicious scripts. The enhanced model performance is further validated by the updated confusion matrix in Table 13, which provides a detailed breakdown of the model's predictive accuracy post-optimization.

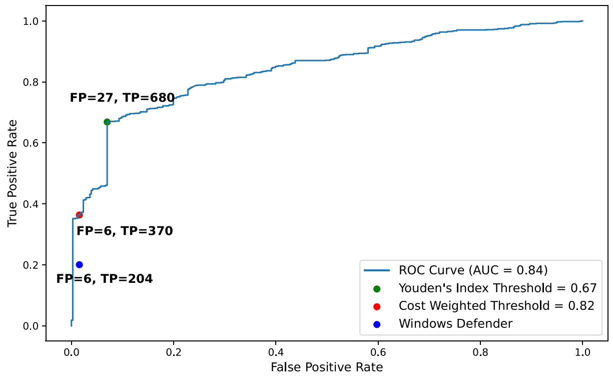


Fig. 8. ROC curve analysis of malware classification performance based on network analysis and L-moments.

7.3. ROC curve and AUC analysis

The Receiver Operating Characteristic (ROC) curve serves as a helpful graphical tool in evaluating the diagnostic capability of binary classifiers, illustrating how their discrimination thresholds impact performance (Swets, 1988). It plots the TPR, also known as sensitivity, against the FPR, or 1 minus specificity, for various threshold settings. The Area Under the Curve (AUC) quantifies the model's overall ability to distinguish between classes, with a higher AUC signifying superior performance (Bradley et al., 1998). As depicted in Fig. 8, the optimized logistic regression model's ROC curve, with an AUC of 0.84, showcases its effective discriminative power in malware classification. That is, if presented with a malicious and a non-malicious script, the model will correctly classify them with a probability of 0.84. Notably, this model optimally minimizes FPs, significantly improving over traditional detection methods like Defender that attempt to minimize FPs.

The performance of Defender does not align with the optimal points on the ROC curve in Fig. 8, suggesting that this method can reduce FPs more effectively. Intriguingly, at certain operating points, this method not only minimizes FPs better than Defender, but also achieves higher TPs simultaneously. This aspect is particularly encouraging, as it indicates that the approach can effectively balance the trade-off between capturing true malware instances and minimizing false alarms. To better illustrate these differences, future iterations of the ROC curve could be enhanced by labeling specific points with their corresponding FP and TP values, providing a quantitative measure of the performance comparison and improvement over traditional methods like Defender.

Further analysis compared this logistic regression model with Microsoft Defender, particularly using a cost-sensitive approach akin to Defender's. In this approach, a FP is deemed more costly than a

FN, aligning with scenarios where the repercussions of false alarms, such as in malware detection, are significant and warrant a higher penalty. The total cost of misclassifications in our model was quantified using a custom cost function, as defined in Eq. (13), where w_0 and w_1 represent the costs for FPs (FP = 1 - specificity) and FNs (FN = 1 - sensitivity), respectively, with each FP incurring a higher cost than a FN (Elkan, 2001). This method emphasizes the importance of accounting for decision costs and uncertainty around the cutoff in diagnostic tests (Skaltsa et al., 2009). The cost function equation reflects this differential weighting, which is crucial in scenarios like malware detection where the consequences of FPs can be more severe than FNs.

$$C = w_0\text{FP} + w_1\text{FN} \tag{13}$$

Optimizing the classification using this cost function resulted in an optimal threshold where the FPR matched Defender's weighted cost of misclassification. The confusion matrix at this threshold, shown in Table 14, indicates a prediction of 380 TNs and 370 TPs, alongside 6 FPs and 647 FNs.

The accuracy of our model at this optimal threshold stands at approximately 0.53, indicating that more than half of the total predictions made by the model were correct. Precision is high at approximately 0.98, suggesting that when the model predicts an instance to be positive, it is likely to be correct. However, recall is notably lower at approximately 0.39, reflecting the model's conservative stance in predicting positive instances, likely influenced by the heavy penalty for FPs that mirrors Defender. This results in an F_1 score of approximately 0.53.

Upon implementing an adjusted threshold to specifically accommodate the cost implications of FPs, it is worth noting that such a framework is pivotal when considering the asymmetric nature of error costs typical in malware detection scenarios. Notably, this approach mirrors real-world operational conditions more closely than traditional evaluation metrics, offering a nuanced comparison to established tools like Microsoft Defender. Particularly, this model's performance in detecting known malware, validated against a dataset that includes a balanced representation of malicious and non-malicious PowerShell scripts, suggests a comparable level of effectiveness and potentially exceeds that of Windows Defender.

8. Conclusion & Future work

This research represents a significant enhancement in malware detection methodologies by integrating network analysis and L-moments within the framework of ASTs. This approach capitalizes on the structural insights of ASTs combined with L-moments and has demonstrated its potential to advance beyond traditional static analysis methods, such as those used by Windows Defender.

The logistic regression model, fitted with L-moments derived from network measures, has shown a promising ability to detect malware previously unknown to the model, maintaining moderate performance. Compared to traditional static detection methods, like those used in Windows Defender, this approach exhibited a reduced rate of FNs, indicating a more effective strategy for classifying malicious code. It is reasonable to conclude that with further refinement and development, the performance of such models can improve over time.

The success of this approach in detecting unknown threats and the methodology for evaluating its effectiveness underscore the potential of advanced statistical and analytical techniques in enhancing malware detection. As the performance of these models improves and the models are employed in defensive systems, malware writers will likely develop new strategies to defeat them, which could be the subject of future research.

Future work

This research represents a foundational exploration into malware detection using real data to tackle genuine threats, highlighting the current framework's potential and preliminary nature. Future efforts should refine the classification methodology to enhance performance significantly (Buczak and Guven, 2016). One promising direction is the adaptation of clustering techniques to ASTs, addressing the inherent challenges of their unique tree-like structures and lack of cycles, which pose a challenge for standard clustering methods (Saxe and Berlin, 2015).

Decomposing ASTs into subtrees for targeted analysis presents an opportunity to delve deeper into the intricacies of code structure (Rusak et al., 2018). By examining these subtrees, which may represent distinct logical code segments, researchers can gain a more granular perspective on the underlying patterns and behaviors. This subdivision could prove crucial in developing clustering techniques designed explicitly for tree-like data structures, thereby unlocking new dimensions in malware detection (Li et al., 2019). The potential for these techniques to utilize the rich structural data within ASTs offers a promising avenue for future investigations.

Moreover, the scope of this research extends beyond the current focus on PowerShell and C# to encompass any binary, script, or library across various Instruction Set Architecturess (ISAs) and programming languages. This broad applicability underscores the potential to adapt to the evolving landscape of malware threats (Aho et al., 2006). As malware authors continuously adapt their strategies in response to advancements in detection methodologies, the field remains a dynamic and challenging arena for research. Future work must navigate these complexities, exploring the full breadth of possibilities that ASTs and related technologies offer for enhancing malware detection capabilities.

Disclaimer

The views expressed in this paper are those of the authors, and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the U.S. Government. This document has been approved for public release; distribution unlimited, case #88ABW-2024-0261.

CRediT authorship contribution statement

Anthony J. Rose: Writing – review & editing, Writing – original draft, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Christine M. Schubert Kabban:** Writing – review & editing, Validation, Supervision, Project administration, Methodology, Investigation. **Scott R. Graham:** Writing – review & editing, Supervision, Resources, Project administration, Funding acquisition. **Wayne C. Henry:** Writing – review & editing, Supervision. **Christopher M. Rondeau:** Writing – review & editing, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

References

Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D., 2006. Compilers: Principles, Techniques, and Tools, second ed. Addison Wesley, ISBN: 0321486811.

Albert, Jean-Marc, 2024. PowerShell AdminScripts. URL https://github.com/wikijm/PowerShell-AdminScripts. (Accessed 20 June 2024).

Andrews, Oliver, 2024. PowerShell WindowsAdmin. URL https://github.com/unkn0wnvariable/PowerShell-WindowsAdmin. (Accessed 20 June 2024).

Bekker, Eugen, Lipkau, Oliver, Deceuninck, Tomas, Heusinger, John, 2024. ConfluencePS. URL https://github.com/AtlassianPS/ConfluencePS. (Accessed 20 June 2024).

Bergstra, James, Bengio, Yoshua, 2012. Random search for hyper-parameter optimization. J. Mach. Learn. Res. (ISSN: 1532-4435) 13 (null), 281–305. http://dx.doi.org/10.5555/2188385.2188395.

Bradley, P., Fayyad, U., Reina, C., 1998. Scaling clustering algorithms to large databases. In: Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining. In: KDD 98, AAAI Press, pp. 9–15. http://dx.doi.org/10.5555/3000292.3000295.

Buczak, A.L., Guven, E., 2016. A survey of data mining and machine learning methods for cyber security intrusion detection. IEEE Commun. Surv. Tutor. 18 (2), 1153–1176. http://dx.doi.org/10.1109/COMST.2015.2494502.

Burian, A., Takala, J., Ylinen, M., 2003. A fixed-point implementation of matrix inversion using cholesky decomposition. In: 2003 46th Midwest Symposium on Circuits and Systems. Vol. 3, pp. 1431–1434. http://dx.doi.org/10.1109/MWSCAS.2003.1562564, Vol. 3.

Chawla, N.V., Bowyer, K.W., Hall, L.O., Kegelmeyer, W.P., 2002. SMOTE: Synthetic minority over-sampling technique. J. Artificial Intelligence Res. (ISSN: 1076-9757) 16, 321–357. http://dx.doi.org/10.1613/jair.953.

Chen, J., Deng, R., 2022. Similarity-based malware classification using graph neural networks. Appl. Sci. (ISSN: 2076-3417) 12 (21), 10837. http://dx.doi.org/10.3390/app122110837.

Concas, G., Marchesi, M., Pinna, S., Serra, N., 2007. Power-laws in a large object-oriented software system. IEEE Trans. Softw. Eng. 33, 687–708. http://dx.doi.org/10.1109/TSE.2007.1019.

Diestel, R., 2017. Graph Theory, fifth ed. Springer Graduate Texts in Mathematics, Springer-Verlag, ISBN: 9783961340057.

Ding, Y., Xia, X., Chen, S., Li, Y., 2018. A malware detection method based on family behavior graph. Comput. Secur. (ISSN: 0167-4048) 73, 73–86. http://dx.doi.org/10.1016/j.cose.2017.10.007.

Elkan, C., 2001. The foundations of cost-sensitive learning. In: Proceedings of the 17th International Joint Conference on Artificial Intelligence - Volume 2. In: IJCAI 01, Vol. 17, Morgan Kaufmann Publishers Inc., ISBN: 1558608125, pp. 973–978. http://dx.doi.org/10.5555/1642194.1642224.

Fallahgoul, H.A., Mancini, L., Stoyanov, S.V., 2023. An L-moment approach for portfolio choice under non-expected utility. In: 31st Australasian Finance and Banking Conference 2018; Proceedings of Paris December 2019 Finance Meeting EUROFIDAI - ESSEC. http://dx.doi.org/10.2139/ssrn.3221471, Swiss Finance Institute Research Paper No. 18-65.

Fang, Y., Zhou, X., Huang, C., 2021. Effective method for detecting malicious PowerShell scripts based on hybrid features. Neurocomputing (ISSN: 0925-2312) 448, 30–39. http://dx.doi.org/10.1016/j.neucom.2021.03.117.

Ferreira, Pedro, Le, Duc C., Zincir-Heywood, Nur, 2019. Exploring feature normalization and temporal information for machine learning based insider threat detection. In: 2019 15th International Conference on Network and Service Management. CNSM, pp. 1–7. http://dx.doi.org/10.23919/CNSM46954.2019.9012708.

Freeman, L.C., 1977. A set of measures of centrality based on betweenness. Sociometry 40 (1), 35–41. http://dx.doi.org/10.2307/3033543.

Freeman, L.C., 1978. Centrality in social networks conceptual clarification. Social Networks (ISSN: 0378-8733) 1 (3), 215–239. http://dx.doi.org/10.1016/0378-8733(78)90021-7.

Ghamrawi, N., McCallum, A., 2005. Collective multi-label classification. In: Proceedings of the 14th ACM International Conference on Information and Knowledge Management. In: CIKM 05, Association for Computing Machinery, New York, NY, USA, ISBN: 1595931406, pp. 195–200. http://dx.doi.org/10.1145/1099554.1099591.

Gill, Andy, 2024. CVE-2020-1350. URL https://github.com/ZephrFish/CVE-2020-1350_HoneyPoC. (Accessed 20 June 2024).

Godbole, S., Sarawagi, S., 2004. Discriminative methods for multi-labeled classification. In: Advances in Knowledge Discovery and Data Mining. Springer Berlin Heidelberg, Berlin, Heidelberg, ISBN: 978-3-540-24775-3, pp. 22–30. http://dx.doi.org/10.1007/978-3-540-24775-3_5.

Guttman, N.B., 1993. The use of L-moments in the determination of regional precipitation climates. J. Clim. 6 (12), 2309–2325. http://dx.doi.org/10.1175/1520-0442(1993)006<2309:TUOLMI>2.0.CO;2.

Han, J., Kamber, M., Pei, J., 2011. Data Mining: Concepts and Techniques (3rd Edition), third ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, ISBN: 9780123814791, http://dx.doi.org/10.1016/C2009-0-61819-5.

Hendler, D., Kels, S., Rubin, A., 2018. Detecting malicious PowerShell commands using deep neural networks. In: Proceedings of the 2018 on Asia Conference on Computer and Communications Security. In: ASIACCS 18, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450355766, pp. 187–197. http://dx.doi.org/10.1145/3196494.3196511.

Hendler, D., Kels, S., Rubin, A., 2020. AMSI-based detection of malicious PowerShell code using contextual embeddings. In: Proceedings of the 15th ACM Asia Conference on Computer and Communications Security. In: ASIA CCS 20, Association for Computing Machinery, New York, NY, USA, pp. 679–693. http://dx.doi.org/10.1145/3320269.3384742.

Hosking, J.R.M., 1990. L-moments: Analysis and estimation of distributions using linear combinations of order statistics. J. R. Stat. Soc. Ser. B Stat. Methodol. (ISSN: 00359246) 52 (1), 105–124. http://dx.doi.org/10.1111/j.2517-6161.1990.tb01775.x.

Hosking, J.R.M., 1997. Fortran routines for use with the method of L-moments version 3.04.

Hosking, J.R.M., Wallis, J.R., 1997. Regional Frequency Analysis: An Approach Based on L-Moments. Cambridge University Press, ISBN: 9780511529443, http://dx.doi.org/10.1017/CBO9780511529443.

Hu, W., Xie, L., Li, L., Yang, X., Yu, Y., Xu, Y., Shi, Z., 2019. MAD-GAN: Multivariate anomaly detection for time series data with generative adversarial networks. In: Artificial Neural Networks and Machine Learning – ICANN 2019: Text and Time Series. Springer International Publishing, ISBN: 978-3-030-30490-4, pp. 703–716. http://dx.doi.org/10.1007/978-3-030-30490-4_56.

Irons, E.T., 1961. A syntax directed compiler for ALGOL 60. Commun. ACM (ISSN: 0001-0782) 4 (1), 51–55. http://dx.doi.org/10.1145/366062.366083.

Johnson, Matthew, 2024. PoshSec. URL https://github.com/PoshSec/PoshSec. (Accessed 20 June 2024).

Kinable, J., Kostakis, O., 2011. Malware classification based on call graph clustering. J. Comput. Virol. (ISSN: 1772-9904) 7 (4), 233–245. http://dx.doi.org/10.1007/s11416-011-0151-y.

Langlois, P., Pinto, A., Hylender, D., Widup, S., 2023. DBIR 2023 data breach investigations report. Verizon Tech. Rep. http://dx.doi.org/10.13140/RG.2.2.32362.70085.

Li, Z., Chen, Q.A., Xiong, C., Chen, Y., Zhu, T., Yang, H., 2019. Effective and light-weight deobfuscation and semantic-aware attack detection for PowerShell scripts. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. In: CCS '19', Association for Computing Machinery, New York, NY, USA, pp. 1831–1847. http://dx.doi.org/10.1145/3319535.3363187.

Lo, Wai Weng, Layeghy, Siamak, Sarhan, Mohanad, Gallagher, Marcus, Portmann, Marius, 2022. Graph neural network-based android malware classification with jumping knowledge. In: 2022 IEEE Conference on Dependable and Secure Computing. DSC, pp. 1–9. http://dx.doi.org/10.1109/DSC54232.2022.9888878.

Maimon, O., Rokach, L., 2010. Data Mining and Knowledge Discovery Handbook, second ed. Springer, ISBN: 9780387098227, http://dx.doi.org/10.1007/978-0-387-09823-4.

Malware Bazaar, 2024. MalwareBazaar database. URL https://bazaar.abuse.ch/. (Accessed 20 June 2024).

Marra, Tiziano, 2024. CVE-2020-0688. URL https://github.com/MrTiz/CVE-2020-0688. (Accessed 20 June 2024).

Microsoft, 2023a. CommandAst class. URL https://learn.microsoft.com/en-us/dotnet/api/system.management.automation.language.commandast?view=powershellsdk-7.3.0. (Accessed 5 May 2023). PowerShell SDK 7.3 | Microsoft Learn.

Microsoft, 2023b. ExpressionAst class. URL https://learn.microsoft.com/en-us/dotnet/api/system.management.automation.language.expressionast?view=powershellsdk-7.2.0. (Accessed 5 May 2023). PowerShell SDK 7.3 | Microsoft Learn.

Microsoft, 2023c. ParameterAst class. URL https://learn.microsoft.com/en-us/dotnet/api/system.management.automation.language.parameterast?view=powershellsdk-7.2.0. (Accessed 5 May 2023). PowerShell SDK 7.3 | Microsoft Learn.

Microsoft, 2023d. ScriptBlockAst class. URL https://learn.microsoft.com/en-us/dotnet/api/system.management.automation.language.scriptblockast?view=powershellsdk-7.3.0. (Accessed 5 May 2023). PowerShell SDK 7.3 | Microsoft Learn.

Microsoft, 2024a. Ast class. URL https://learn.microsoft.com/en-us/dotnet/api/system.management.automation.language.ast?view=powershellsdk-7.4.0. (Accessed 18 August 2024).

Microsoft, 2024b. Parser class. URL https://learn.microsoft.com/en-us/dotnet/api/system.management.automation.language.ast?view=powershellsdk-7.4.0. (Accessed 18 August 2024).

Microsoft, 2024c. PowerShell documentation - PowerShell. URL https://docs.microsoft.com/en-us/powershell/. (Accessed 12 February 2024).

Mimura, M., Tajiri, Y., 2021. Static detection of malicious PowerShell based on word embeddings. Internet Things (ISSN: 2542-6605) 15, 1–13. http://dx.doi.org/10.1016/j.iot.2021.100404.

Mittal, Nikhil, 2024. Nishang. URL https://github.com/samratashok/nishang. (Accessed 20 June 2024).

MrPowerScripts, 2024. PowerScripts. URL https://github.com/MrPowerScripts/PowerScripts. (Accessed 20 June 2024).

Nettitude, 2024. PoshC2. URL https://github.com/nettitude/PoshC2/. (Accessed 20 June 2024).

Ng, Andrew Y., 2004. Feature selection, L1 vs. L2 regularization, and rotational invariance. In: Proceedings of the Twenty-First International Conference on Machine Learning. ICML '04, Association for Computing Machinery, New York, NY, USA, ISBN: 1581138385, p. 78. http://dx.doi.org/10.1145/1015330.1015435.

Palo Alto - Unit42, 2024. PSencmds. URL https://github.com/pan-unit42/iocs/tree/master/psencmds. (Accessed 20 June 2024).

Pascanu, R., Stokes, J.W., Sanossian, H., Marinescu, M., Thomas, A., 2015. Malware classification with recurrent networks. In: 2015 IEEE International Conference on Acoustics, Speech and Signal Processing. ICASSP, IEEE, pp. 1916–1920. http://dx.doi.org/10.1109/ICASSP.2015.7178304.

Pogonin, D., Korkin, I., 2022. Microsoft defender will be defended: MemoryRanger prevents blinding windows AV. In: The 15th Annual ADFSL Conference on Digital Forensics, Security and Law. http://dx.doi.org/10.48550/arXiv.2210.02821.

PowerShell Team, 2024. PowerShell. URL https://github.com/PowerShell/PowerShell. (Accessed 20 June 2024).

Rapid7, 2024. Metasploit framework. URL https://github.com/rapid7/metasploit-framework. (Accessed 20 June 2024).

Rusak, G., Al-Dujaili, A., O'Reilly, U.M., 2018. AST-based deep learning for detecting malicious PowerShell. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. In: CCS 18, ACM, http://dx.doi.org/10.1145/3243734.3278496.

Sabidussi, G., 1966. The centrality index of a graph. Psychometrika 31 (4), 581–603. http://dx.doi.org/10.1007/BF02289527.

Saxe, J., Berlin, K., 2015. Deep neural network based malware detection using two dimensional binary program features. In: 2015 10th International Conference on Malicious and Unwanted Software. MALWARE, pp. 11–20. http://dx.doi.org/10.1109/MALWARE.2015.7413680.

Security, B.C., 2024. Empire. URL https://github.com/BC-SECURITY/Empire. (Accessed 20 June 2024).

Skaltsa, K., Jover, L., Carrasco, J.L., 2009. Diagnostic tests optimum threshold: Accounting for decision costs and uncertainty around the cut-off. In: Proceedings of the 22nd Panhellenic Conference on Statistics. pp. 333–340.

Song, J., Kim, J., Choi, S., Kim, J., Kim, I., 2021. Evaluations of AI-based malicious PowerShell detection with feature optimizations. ETRI J. 43 (3), 549–560. http://dx.doi.org/10.4218/etrij.2020-0215.

Spelmen, Vimalraj S., Porkodi, R., 2018. A review on handling imbalanced data. In: 2018 International Conference on Current Trends Towards Converging Technologies. ICCTCT, pp. 1–11. http://dx.doi.org/10.1109/ICCTCT.2018.8551020.

Splunk, 2024. Splunk reskit PowerShell. URL https://github.com/nkasco/splunk-reskit-powershell. (Accessed 20 June 2024).

Swets, J.A., 1988. Measuring the accuracy of diagnostic systems. Science 240 (4857), 1285–1293. http://dx.doi.org/10.1126/science.3287615.

Tsai, M.H., Lin, C.C., He, Z.G., Yang, W.C., Lei, C.L., 2023. PowerDP: De-obfuscating and profiling malicious PowerShell commands with multi-label classifiers. IEEE Access 11, 256–270. http://dx.doi.org/10.1109/ACCESS.2022.3232505.

Vargo, E., Pasupathy, R., Leemis, L., 2010. Moment-ratio diagrams for univariate distributions. J. Qual. Technol. 42, http://dx.doi.org/10.1080/00224065.2010.11917824.

VMware, 2024. PowerNSX. URL https://github.com/vmware-archive/powernsx. (Accessed 20 June 2024).

Youden, W.J., 1950. Index for rating diagnostic tests. Cancer 3 (1), 32–35. http://dx.doi.org/10.1002/1097-0142.

Zhu, S., Ji, X., Xu, W., Gong, Y., 2005. Multi-labelled classification using maximum entropy method. In: Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. In: SIGIR 05, Association for Computing Machinery, New York, NY, USA, ISBN: 1595930345, pp. 274–281. http://dx.doi.org/10.1145/1076034.1076082.

Anthony Rose is a Doctoral Student at the Air Force Institute of Technology. His research interests focus on various facets of cybersecurity, exploring the critical interplay between malware, operating system vulnerabilities, and the measures necessary to defend against evolving cyber threats. Specific areas of interest include computer security, malware detection and development, operating system vulnerabilities, and defensive mechanisms against ongoing cyber threats.

Dr. Scott Graham is a Professor of Computer Engineering at the Air Force Institute of Technology. His research interests center on cyber physical systems, looking at the intersection between real physical systems and the computers that control them. Specific areas of interest include cyber physical systems security, computer architecture, embedded computing, computer communication networks, critical infrastructure protection, and vehicular cyber security.

Dr. Christine Schubert Kabban received the B.S. degree in Mathematics from the University of Dayton, M.S. in Applied Statistics from Wright State University and a Ph.D. in Applied Mathematics from the Air Force Institute of Technology (AFIT). She has been researching and practicing statistics for over 20 years in clinical, engineering, and statistical fields and is currently a Professor of Statistics in the Department of Mathematics and Statistics in the Graduate School of Engineering and Management at the Air Force Institute of Technology. Her current work focuses on applications to structural health monitoring, target detection, classification methods, and autonomous systems and networks with hierarchical and complex multi-dimensional data in addition to natural language processing.

Wayne C. Henry is an Assistant Professor of Electrical Engineering at the Air Force Institute of Technology (AFIT), Wright-Patterson AFB, Ohio USA. He received a B.S. in Computer Engineering from Pennsylvania State University in 2004, and both an M.S. in Computer Engineering in 2011 and a Ph.D. in 2020 from AFIT. Lt Col Henry's research interests include space system cyber security, malware analysis, network defense technologies, offensive security, human–machine teaming, and information visualizations.

Christopher M. Rondeau is an Assistant Professor of Electrical Engineering at the Air Force Institute of Technology (AFIT), Wright-Patterson AFB, Ohio USA. He received a B.S. in Electrical Engineering from the United States Air Force Academy in 2003, and both an M.S. in Electrical Engineering in 2010 and a Ph.D. in 2020 from AFIT. Specific research interests include cyber physical systems, RF Fingerprinting, Physical Layer defense and exploitation, machine learning, test technique development, and critical infrastructure security.