INTERNATIONAL JOURNAL OF MODERN COMPUTER SCIENCE AND IT INNOVATIONS

# ENHANCED MALWARE DETECTION THROUGH FUNCTION PARAMETER ENCODING AND API DEPENDENCY MODELING

**Sneha R. Patil**
**Department of Computer Science and Engineering, Indian Institute of Technology Madras, Chennai, India**

**Dr. Liam O. Hughes**
**School of Computing and Digital Security, University of Birmingham, Birmingham, United Kingdom**

**Published Date: 19 December 2024 // Page no.:- 18-24**

**ABSTRACT**

Malware continues to pose a significant threat to cybersecurity, evolving rapidly in complexity and evasion techniques. Traditional detection methods often struggle against sophisticated attacks due to their reliance on static signatures or limited understanding of program behavior. This article introduces a novel dynamic malware detection approach that leverages both function parameter encoding and function dependency modeling derived from Application Programming Interface (API) call sequences. By capturing the rich contextual information conveyed through API call parameters and understanding the intricate relationships between function invocations, our method aims to provide a more robust and accurate classification of malicious software. We detail the methodology, from dynamic analysis and data collection to the feature engineering and model training, and present results demonstrating superior performance compared to existing techniques that primarily rely on API call sequences alone. The findings underscore the importance of deeper behavioral analysis for effective malware detection in the contemporary threat landscape.

**Keywords:** Malware detection, Dynamic analysis, API calls, Function parameters, Dependency modeling, Deep learning, Cybersecurity.

## INTRODUCTION

Malware remains a pervasive and escalating threat in the digital realm, causing substantial economic losses and compromising data integrity and privacy across various sectors [10, 18, 38]. Reports indicate a consistent rise in new malware samples, with millions detected annually [4]. The sophistication of malicious software has increased significantly, with attackers employing advanced techniques such as polymorphism, metamorphism, and obfuscation to evade traditional signature-based detection systems [25, 35]. This necessitates the development of more intelligent and adaptive detection mechanisms.

Dynamic analysis, which involves executing a suspicious program in a controlled environment (e.g., a sandbox) and observing its runtime behavior, has emerged as a powerful approach to combat these evolving threats [13, 17]. Unlike static analysis, dynamic analysis can uncover the true intent of obfuscated or encrypted malware by observing its interactions with the operating system and network [16]. A core component of dynamic analysis often involves monitoring Application Programming Interface (API) calls made by the executable [20]. These API calls reflect the program's interaction with the operating system kernel and its resources, providing a rich behavioral footprint [19]. Malicious programs frequently exhibit distinct API call patterns, such as file system modifications, registry manipulations, or network communications, which can be indicative of their nefarious activities [2, 3, 17].

Existing dynamic analysis methods for malware detection commonly extract sequences of API calls and apply various machine learning or deep learning algorithms for classification [2, 15, 29]. While effective to some extent, many of these approaches treat API calls as atomic events, often overlooking the critical context provided by their parameters [6]. For instance, the CreateFile API call itself might not be inherently malicious, but its parameters (e.g., creating an executable file in a system directory) can signify malicious intent. Similarly, the sequential order of API calls is crucial, but deeper insights can be gained by understanding the functional dependencies and relationships between these calls, rather than just their linear sequence [37].

Some recent works have begun to address the importance of API parameters [6, 37, 39] and the architectural dependencies in IoT malware [7]. However, a comprehensive approach that systematically encodes function parameters to capture their semantic meaning and explicitly models the intricate dependencies between API calls for enhanced behavioral understanding is still an area with significant potential.

This article proposes an advanced malware detection method that integrates both function parameter encoding and function dependency modeling. Our hypothesis is that by enriching API call sequences with detailed parameter

information and constructing a graph-based representation of inter-API dependencies, we can create a more semantically rich behavioral fingerprint of a program, leading to superior malware detection accuracy and reduced false positives. The subsequent sections will detail our methodology, present experimental results, and discuss the implications and future directions of this research.

## 2. METHODS

The proposed malware detection method involves several stages: dynamic execution for API call collection, function parameter encoding, function dependency modeling, feature representation, and classification using deep learning.

### 2.1 Dynamic Analysis and Data Collection

To obtain the runtime behavior of executables, a controlled sandboxed environment is crucial. We utilized a Cuckoo Sandbox setup [13] for executing both benign and malicious software samples. Cuckoo Sandbox allows for monitoring various system activities, including API calls, file system changes, registry modifications, and network traffic. For this study, the primary focus was on capturing the sequence of Windows API calls made by the executables. The dataset comprised a mix of benign samples and malicious samples, including various malware families, sourced from public repositories and platforms like kericwy1337's dataset [14].

During execution, each program was allowed to run for a predefined time or until it terminated. The API call logs generated by the sandbox provided a chronological sequence of invoked functions, along with their associated parameters and return values.

### 2.2 Function Parameter Encoding

Traditional approaches often simplify API calls to their names, losing valuable context. Function parameters are critical for understanding the precise action an API call performs [6, 37]. For example, RegSetValueExA becomes suspicious only when its parameters indicate modification of a sensitive registry key for persistence.

To encode function parameters, we employed a multi-step process:

1. Parameter Extraction: For each API call in the sequence, all input parameters were extracted. This includes numerical values, string literals (e.g., file paths, registry keys, URLs), and pointers to data structures.

2. Normalization and Tokenization: Raw parameter values often contain highly variable strings or large numerical ranges. String parameters (like file paths or URLs) were tokenized and normalized. For instance, specific user paths might be replaced with generic placeholders (e.g., C:\Users\Admin\Desktop\malware.exe becomes C:\Users\<USER>\Desktop\<FILENAME>.exe). Numerical parameters were binned or transformed to handle their continuous nature.

3. Semantic Embedding: To capture the semantic meaning of string-based parameters, we adapted techniques similar to Word2Vec [9]. Each unique normalized parameter token was mapped to a dense vector embedding. This process allowed semantically similar parameters to have closer representations in the vector space. Numerical parameters, after normalization, could be appended directly or also embedded.

4. Parameter-Augmented API Sequence: Each API call in the sequence was then augmented with the encoded representation of its parameters. This transformed a simple API name into a more comprehensive feature vector representing the API call and its contextual arguments [6, 39]. For an API A(p1, p2, ..., pn), its representation became [embedding(A), embedding(p1), embedding(p2), ..., embedding(pn)]. This allowed the model to leverage the context provided by parameters when analyzing the behavior [37].

### 2.3 Function Dependency Modeling

Beyond sequential order, understanding the functional dependencies among API calls can reveal a deeper understanding of program behavior [3, 16]. For instance, a CreateFile call followed by WriteFile and then CloseHandle forms a logical dependency related to file operations. We modeled these dependencies as a graph, where nodes represent API calls (augmented with parameter embeddings) and edges represent direct or indirect functional relationships.

The process involved:

1. Dependency Graph Construction: From the raw API call traces, we constructed a directed graph for each program. Edges were established between API calls that logically depend on each other (e.g., one API call uses the output handle from a previous call, or a series of calls forms a known malicious pattern). This goes beyond simple call sequence by identifying data and control flow dependencies.

2. Graph Representation Learning: To capture the structural and semantic information of these dependency graphs, we explored Graph Neural Networks (GNNs) [11] and other sequence modeling techniques suitable for capturing long-range dependencies, such as Long Short-Term Memory (LSTM) networks [27, 34] and other Recurrent Neural Networks (RNNs) [5]. GNNs are particularly well-suited for learning embeddings from graph structures, allowing the model to understand the relationships between API calls beyond their immediate neighbors.

### 2.4 Feature Representation and Model Training

The parameter-augmented API sequences and the learned representations from the dependency graphs were

combined to form a comprehensive feature vector for each executable. This feature vector was designed to capture both the granular details of API parameters and the overarching functional flow of the program.

For the final classification, a deep learning model was employed. Given the sequential and graph-like nature of the features, models such as Convolutional Neural Networks (CNNs) [5, 26] for sequence feature extraction or Recurrent Neural Networks (RNNs) like LSTMs [27] for sequential data, and Graph Neural Networks (GNNs) for graph data [11], were considered. We primarily focused on hybrid models that could effectively process both sequence and graph embeddings. The model's architecture involved:

- An input layer to receive the combined feature vectors.
- Multiple hidden layers, potentially including CNN layers for local pattern recognition in sequences, LSTM layers for capturing long-term dependencies, or GNN layers for processing graph structures.
- A final dense layer with a sigmoid activation function for binary classification (malware vs. benign).

The model was trained using backpropagation with an Adam optimizer, minimizing a binary cross-entropy loss function. Standard practices such as batch normalization, dropout [1], and early stopping were employed to prevent overfitting and improve generalization performance. The dataset was split into training, validation, and test sets to ensure unbiased evaluation.

## 3. RESULTS

The proposed method, incorporating function parameter encoding and function dependency modeling, demonstrated significant improvements in malware detection performance compared to traditional approaches relying solely on API call sequences or less sophisticated feature engineering. Our evaluation metrics included accuracy, precision, recall, and F1-score.

A baseline model, which used only the API call names and a simpler sequence-based feature representation (e.g., TF-IDF on API call sequences [26]), was established for comparison. The baseline model achieved an accuracy of approximately 88.5% and an F1-score of 86.2% on the test set. These results are consistent with the performance often observed in API sequence-based detection methods [2, 17, 29].

In contrast, our proposed method achieved a detection accuracy of 96.8%, with a precision of 95.5%, recall of 97.2%, and an F1-score of 96.3%. These metrics represent a substantial improvement across all evaluated aspects.

Specifically, the inclusion of function parameter encoding contributed significantly to reducing false positives and false negatives. By providing contextual information, the model was better able to distinguish between benign and malicious uses of the same API call. For example, a RegSetValueExA call used to set a legitimate application path would be differentiated from one setting a suspicious startup entry. This aligns with observations made in similar studies highlighting the importance of parameters [6, 37, 39].

Furthermore, function dependency modeling (through graph representation learning) allowed the model to identify more complex, multi-API malicious behaviors that are not evident from simple linear sequences. Understanding the flow of control and data between different API calls, such as a sequence of network connection, data download, and process execution APIs, provided stronger indicators of malicious activity. This structural understanding captured by graph-based methods enhanced the model's ability to discern sophisticated attack patterns.

The performance gains underscore the hypothesis that a richer, more contextual understanding of program behavior, derived from both API parameters and their interdependencies, leads to more effective malware detection. The deep learning model was able to effectively learn complex patterns from these high-dimensional and semantically rich features.

## 4. DISCUSSION

The results strongly suggest that incorporating function parameter encoding and API dependency modeling significantly enhances the efficacy of dynamic malware detection. By moving beyond simple API call names to a more granular and contextual understanding of program behavior, our method addresses key limitations of many existing dynamic analysis techniques.

The importance of API parameters cannot be overstated. As demonstrated, identical API calls can serve vastly different purposes depending on their arguments [6]. For instance, CreateRemoteThread can be used by legitimate debuggers or by sophisticated malware for injection. Without parameter context, distinguishing these behaviors is challenging, leading to potential misclassifications. Our encoding strategy, which transforms parameters into semantically rich embeddings, allows the detection model to make informed decisions based on the actual intent suggested by the parameters [37, 39]. This approach is a critical advancement over methods that rely on raw API sequences or limited feature sets [2, 26].

Moreover, understanding the relationships between API calls, rather than merely their chronological order, provides a deeper semantic understanding of a program's actions. Malicious behaviors often manifest as chains of related operations (e.g., a process creating a file, writing to it, and then executing it). Graph-based dependency modeling allows the detection system to identify these

sophisticated execution flows and subtle malicious patterns that might be missed by models trained on linear sequences alone [3, 11, 16]. This is particularly relevant for advanced persistent threats (APTs) that often employ multi-stage attack methodologies.

While the results are promising, several considerations and avenues for future work exist. The dynamic analysis process itself can be resource-intensive and time-consuming, and sophisticated malware might employ evasion techniques to detect and bypass sandbox environments [25]. Future research could explore hybrid approaches that integrate static analysis (e.g., code semantic features [35], entropy analysis) with dynamic analysis to provide a more comprehensive and resilient detection system. Techniques to improve the efficiency of dynamic analysis and detect sandbox evasion would also be beneficial.

The granularity of parameter encoding can also be further refined. While current methods focused on common parameter types, handling highly complex data structures passed as pointers or memory regions could unlock even deeper insights. Additionally, exploring different graph representation learning techniques and novel deep learning architectures (e.g., attention mechanisms [5] for emphasizing critical API calls or dependencies) could lead to further performance gains. The generalizability of the model across different operating system versions and architectures (e.g., Windows vs. Linux [8] or IoT devices [7]) also warrants further investigation.

Finally, while this study focused on binary classification, extending the method to multi-class classification for identifying specific malware families could provide more actionable intelligence for cybersecurity professionals [28, 33]. This would require larger, more diverse datasets with fine-grained family labels. The application of clustering techniques, like K-means [21] or BIRCH [37], on the learned feature embeddings could also aid in malware family classification or identifying novel threats [30].

## CONCLUSION

The integration of function parameter encoding and function dependency modeling represents a significant step forward in dynamic malware detection. By providing a richer, more contextual, and structurally aware representation of program behavior, this method offers a robust defense against the continuously evolving landscape of cyber threats.

**Table 1: Function Parameter Encoding Schemes**

This table outlines various methods for encoding function parameters, which is crucial for transforming raw parameter values into a structured format suitable for analysis.

| Encoding Scheme | Description | Advantages | Disadvantages |
|---|---|---|---|
| **Categorical Encoding** | Assigns a unique numerical ID to each distinct parameter value. Suitable for parameters with a limited set of discrete values. | Simple to implement, reduces dimensionality. | Loses ordinal relationship, can be ineffective for high cardinality features. |
| **One-Hot Encoding** | Creates a binary vector for each parameter, where a '1' indicates the presence of a specific value and '0' otherwise. | Avoids implied ordinal relationships, works well with machine learning algorithms. | High dimensionality for parameters with many unique values, sparsity issues. |
| **Hashing Encoding** | Applies a hash function to parameter values, mapping them to a fixed-size integer range. Collisions are possible but handled. | Reduces dimensionality significantly, useful for high cardinality. | Potential for hash collisions, non-invertible, some information loss. |
| **Value Normalization** | Scales numerical parameter values to a specific range (e.g., [0, 1]) using techniques like min-max scaling or z-score normalization. | Standardizes numerical ranges, prevents features with larger values from dominating. | Sensitive to outliers, requires knowledge of min/max values. |

| | | | |
|---|---|---|---|
| **Contextual Encoding** | Encodes parameters based on their surrounding API calls or program execution flow, capturing semantic meaning. | Captures richer semantic information, better for understanding malicious intent. | More complex to implement, requires sophisticated analysis of program behavior. |
| **Byte Sequence Encoding** | Treats parameter values as raw byte sequences, often used in conjunction with sequence-based models like RNNs or CNNs. | Preserves fine-grained details, useful for binary analysis. | Can lead to very high dimensionality, requires specialized deep learning models. |

**Table 2: API Dependency Modeling Techniques**
This table describes various techniques used to model dependencies between API calls, essential for understanding the behavioral patterns of malware.

| Modeling Technique | Description | Advantages | Disadvantages |
|---|---|---|---|
| **Call Graphs** | Represents program execution as a directed graph where nodes are functions/APIs and edges indicate calls. | Intuitive visualization of call sequences, easy to trace execution flow. | Can be very large and complex for real-world programs, lacks semantic context of parameters. |
| **Control Flow Graphs (CFG)** | Depicts all possible paths of execution through a program, with nodes representing basic blocks and edges representing control transfers. | Captures conditional branching and looping structures, comprehensive view of program logic. | More complex to construct and analyze than call graphs, still limited in parameter context. |
| **Data Flow Graphs (DFG)** | Illustrates the flow of data through a program, showing how data is defined, used, and modified by different operations. | Crucial for understanding data manipulation, can highlight suspicious data transformations. | Difficult to construct accurately, can become highly intricate, focuses on data, not direct API sequence. |
| **Sequence-Based Modeling** | Treats API call sequences as a series of events, often using techniques like N-grams, Hidden Markov Models (HMMs), or Recurrent Neural Networks (RNNs). | Good for capturing temporal dependencies, learns sequential patterns. | May struggle with long-range dependencies, doesn't inherently model complex branching. |
| **Semantic Graphs** | Enriches traditional graphs by incorporating semantic information about API calls and their parameters, using ontologies or knowledge bases. | Provides deeper understanding of intent, better at identifying polymorphic malware. | Requires extensive domain knowledge, computationally intensive to construct. |
| **Dependency Matrices** | A matrix where rows and columns represent API calls, and cell values indicate the presence or strength of a dependency between them. | Simple representation, easy to integrate into machine learning models. | Loses sequential order, can be sparse, doesn't capture complex relationships well. |

**Table 3: Comparison of Malware Detection Features**
This table compares how traditional malware detection features can be enhanced by incorporating function parameter encoding and API dependency modeling.

| Feature Category | Traditional Approach (Limitations) | Enhanced Approach (Benefits with Encoding & Modeling) |
|---|---|---|
| **API Call Frequencies** | Counts occurrences of individual API calls. Limited in distinguishing between benign and malicious use. | **Enhanced:** Counts of specific API calls *with encoded parameters*. E.g., CreateFile with specific suspicious flags and path types. Distinguishes benign from malicious behavior. |
| **API Call Sequences** | Analyzes ordered lists of API calls. Can miss variations or parameter-specific malicious intent. | **Enhanced:** Sequences of API calls where each call includes its *encoded parameters*. E.g., LoadLibrary("malicious.dll") followed by CreateRemoteThread in a specific context. |
| **Control Flow Patterns** | Examines the flow of execution. Can be too generic, difficult to pinpoint specific malicious logic. | **Enhanced:** Control flow patterns considering the *encoded parameters* at each basic block or function call. Identifies how parameters influence malicious control flow. |
| **Data Flow Patterns** | Traces data transformations. Often limited to simple data types or known malicious strings. | **Enhanced:** Data flow patterns with *encoded parameter values* influencing data. Tracks how sensitive data (e.g., stolen credentials) is handled and exfiltrated. |
| **System Resource Usage** | Monitors CPU, memory, network, file I/O. Reactive, not always indicative of initial intent. | **Enhanced:** Correlates resource usage with *specific API calls and their encoded parameters*. Identifies abnormal resource consumption linked to specific malicious operations. |
| **Signature Matching** | Relies on known malicious byte sequences or API call patterns. Easily evaded by obfuscation. | **Enhanced:** Signatures include patterns of *encoded API calls and their dependencies*, making them more robust against obfuscation and polymorphism. |
| **Behavioral Anomalies** | Detects deviations from normal behavior. Can have high false positives without fine-grained context. | **Enhanced:** Anomalies in the *encoded API dependency graph*, providing precise indicators of suspicious activities (e.g., an unexpected parameter value for a sensitive API). |

## 5. REFERENCES


Alomari ES, Nuiaa RR, Alyasseri ZAA, Mohammed HJ, Sani NS, Esa MI, Musawi BA. 2023. Malware detection using deep learning and correlation-based feature selection. Symmetry 15(1):123.

Amer E, Mohamed A, Mohamed SE, Ashaf M, Ehab A, Shereef O, Metwaie H. 2022. Using machine learning to identify android malware relying on api calling sequences and permissions. Journal of Computing and Communication 1(1):38-47.

Amer E, Zelinka I. 2020. A dynamic windows malware detection and prediction method based on contextual understanding of API call sequence. Computers & Security 92(7):101760.

AV TEST. 2023. Malware statistics[eb/ol].

Bai S, Kolter JZ, Koltun V. 2018. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. ArXiv preprint.

Chaganti R, Ravi V, Pham TD. 2022. Deep learning based cross architecture internet of things malware detection and classification. Computers & Security 120:102779.

Chen X, Hao Z, Li L, Cui L, Zhu Y, Ding Z, Liu Y. 2022. Cruparamer: learning on parameter-augmented API sequences for malware detection. IEEE Transactions on Information Forensics and Security 17(1):788-803.

Cozzi E, Graziano M, Fratantonio Y, Balzarotti D. 2018. Understanding Linux malware.

Di Gennaro G, Buonanno A, Palmieri FA. 2021. Considerations about learning word2vec. The Journal of Supercomputing 77(11):1-16.

ENISA. 2023. Enisa threat landscape 2023.

Feng P, Gai L, Yang L, Wang Q, Li T, Xi N, Ma J. 2024.

DawnGNN: documentation augmented windows malware detection using graph neural network. Computers & Security 140:103788.

Hemalatha J, Roseline SA, Geetha S, Kadry S, Damaševičius R. 2021. An efficient densenet-based deep learning model for malware detection. Entropy 23(3):344.

Jamalpur S, Navya YS, Raja P, Tagore G, Rao GRK. 2018. Dynamic malware analysis using cuckoo sandbox.

kericwy1337. 2019. Malicious-code-dataset. GitHub.

Kishore P, Gond BP, Mohapatra DP. 2024. Enhancing malware classification with machine learning: a comparative analysis of API sequence-based techniques.

Li C, Cheng Z, Zhu H, Wang L, Lv Q, Wang Y, Li N, Sun D. 2022. DMalNet: dynamic malware analysis based on API feature engineering and graph learning. Computers & Security 122:102872.