Published by :
https://www.ijert.org/
An International Peer-Reviewed Journal

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

AIRE: An AI-Assisted Semantic Framework for Reverse Engineering of VB.NET Assemblies

Dr. Joseph Raymond V
Networking and Communications (NWC)
SRM Institute of Science and Technology
Chennai, India

Rohith Mathew
Networking and Communications (NWC)
SRM Institute of Science and Technology
Chennai, India

Ishit Jain
Networking and Communications (NWC)
SRM Institute of Science and Technology
Chennai, India

Joel Jacob
Networking and Communications (NWC)
SRM Institute of Science and Technology
Chennai, India

***Abstract*—Reverse engineering of binaries is a critical task in the field of cybersecurity, malware detection, and vulnerability detection. Current reverse engineering tools offer limited capabilities in terms of source code reconstruction, as most of the tools offer low-level disassembly and decompilation of the binary code, requiring human interpretation to understand the actual functionality of the code. The authors of the paper propose a new AI-based semantic reverse engineering framework called AIRE, which can analyze VB.NET Windows x64 assemblies at the method level using intermediate language extraction and AI-based semantic reasoning to analyze the functionality of the code, stack usage, and detection of security vulnerabilities in the code. The authors have also performed experiments to show the effectiveness of the proposed approach in terms of reduced human analysis time.**

***Index Terms*—Reverse Engineering, Artificial Intelligence, VB.NET, Intermediate Language, Static Analysis, Semantic Reasoning, Cybersecurity**

I. Introduction

Reverse engineering is a vital process used in the fields of cybersecurity, digital forensics, malware detection, and software maintenance. In many practical scenarios, reverse engineering plays a vital role in analyzing compiled binaries without the presence of the source code. These scenarios may be restricted by proprietary software, outdated systems, and malware. Reverse engineering tools like disassemblers and decompilers offer a detailed analysis of the instruc- tional code present in the executable file. However, these representations of the software code often require signifi- cant interpretation to comprehend the actual function of the software. In the context of the .NET platform, the com- piled software code is stored as Common Intermediate Language (CIL) code, which is often referred to as the Intermediate Language (IL) code. Although the IL dis- assembler provides a detailed analysis of the instructions present within the methods, this analysis is based on the syntactic structure of the code.

To address the limitations mentioned above, this paper proposes AIRE, an AI-assisted semantic reverse engineering framework for VB.NET Windows X64 Assemblies. AIRE leverages the strength of AI techniques combined with the power of structure in IL extraction to reason and determine the purpose of the method, inputs and outputs, stack information, and security issues associated with the code. Unlike other reverse engi- neering techniques that aim to recreate the source code, AIRE emphasizes the generation of a structured semantic summary in machine-readable JSON format.

II. Related Work

Traditional reverse engineering tools like IDA Pro and Ghidra offer sophisticated capabilities for disassembly and decompila- tion. However, they mainly emphasize visualization at the instruc- tion level. They also involve manual semantic reasoning [1].

In the context of a managed platform, reverse engineering tools like ILSpy and dnSpy offer the ability to examine .NET assemblies. Although these tools offer a high-level code construct from IL code, they do not offer automated semantic reasoning for the intent of a particular method [2].

Recent research focuses on the application of machine learning for malware analysis. Ucci et al. present a comprehensive survey of the application of AI for malware detection, emphasizing the challenges associated with semantic feature extraction [3]. Another study focuses on the application of a deep learning technique like Asm2Vec to detect semantic similarity at the instruction level [4].

Unlike other reverse engineering tools, AIRE offers the capability to extract IL code using structured techniques coupled with the application of AI to offer semantic insights.

III. System Architecture

The AIRE framework has been designed with the aim of providing a modular and extensible architecture that combines the benefits of static IL extraction with the capabilities of AI-based semantic reasoning. The framework has been designed with the aim of providing a layered processing pipeline,

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

which ensures that the processing of the assembly follows a structured approach. The overall workflow can be summarized as follows: assembly ingestion, IL parsing, structured serialization, semantic inference, and finally, visualization.The overall architecture of the AIRE framework is illustrated in Fig. 1.

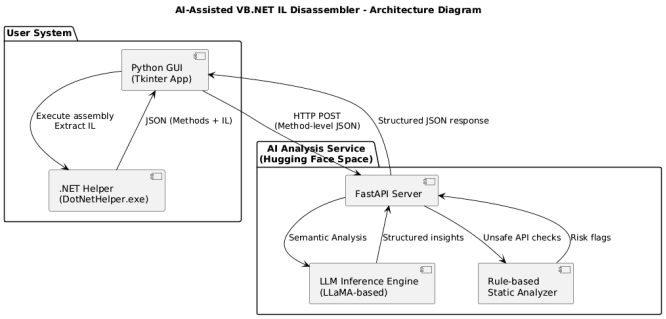

Fig. 1. AI-Assisted VB.NET IL Disassembler Architecture showing interaction between the Python GUI, .NET helper module, and AI analysis service.

AIRE consists of four primary modules:

A. IL Extraction Engine

The IL Extraction Engine is the main static analysis module used for parsing the VB.NET Windows x64 assemblies. It works directly on compiled executable (.exe) and dynamic link library (.dll) files without the need for source code.

The process begins with loading and validating the assembly, which ensures that the provided binary is a legitimate .NET assembly. After validation, the system locates all defined classes within the program by enumerating namespaces and types. The module then extracts method metadata, acquiring details like the method's name, its complete signature, access modifiers, attributes, parameter types, and return types. Following this, the system inspects the method body to analyze its internal structure. This phase involves retrieving Intermediate Language (IL) instructions, which offer crucial information including opcodes, operands, offsets, and stack effects. Furthermore, it identifies local variable declarations and detects exception handling regions, such as try/catch blocks.

The engine provides structural completeness with the ordering of the instructions and the level of granularity at the method level. Unlike an ordinary decompiler, this module does not attempt to produce high-level source code. It concentrates on producing low-level code with accuracy.

B. JSON Structuring Module

The JSON Structuring Module's function is to convert the extracted Intermediate Language (IL) instructions and their related metadata into a structured, machine-readable format. This conversion is essential because it allows the data to be easily processed by AI models in later stages of the system. The module organizes the information in a hierarchical structure that follows the order :

$$\text{Assembly} \rightarrow \text{Types} \rightarrow \text{Methods} \rightarrow \text{IL Instructions}$$

Each JSON object stores important details such as assembly metadata (including name, version, and architecture), type-level information, method-level structural data, and the ordered sequence of IL instructions. In addition, it includes stack-related metadata and information about exception handling blocks within the code. Representing the data in this standardized JSON format ensures consistency and makes the framework easier to extend in the future. It also makes things easier for semantic analysis engines, knowledge graph systems, and rule-based analysis tools. By translating program structure into JSON, the AIRE framework allows for seamless interaction between various analysis components and outside tools.

C. AI Semantic Analysis Engine

The role of the AI semantic analysis engine is to convert structured IL data into high-level semantic form. The module utilizes an instruction-tuned large language model to carry out contextual reasoning on method-level representations.

The engine performs:

- Function intent inference
- Input-output behavior prediction
- Stack usage analysis
- Control flow interpretation
- Identification of suspicious API usage
- Detection of potential security vulnerabilities

Instead of generating reconstructed source codes, the module developed by the AI generates structured semantic summaries that contain the description of the function's purpose, the explanation of the function's existence, and the explanation of the risk posed by the function. Such a generated output can be easily understood and interpreted by machines, which can be used to create documents, detect vulnerabilities, and offer educational assistance.

Through the integration of the reasoning power of the AI, AIRE fills the gap between disassembly and semantic analysis.

D. User Interface Layer

The User Interface Layer is an interactive environment for visualizing IL data and semantic explanations. The User Interface Layer is intended to be usable for educational and research purposes.

The User Interface Layer allows:

- Assembly upload and parsing
- Method-level browsing
- Side-by-side IL and semantic explanation view
- Vulnerability flag visualization
- Structured report export

The interface reduces the cognitive load for users, as it shows them the semantic summaries along with the raw IL instructions, which helps them understand the program's behavior without having to interpret the low-level instructions. This enhances the tool's utility for both students studying reverse engineering and researchers engaged in security investigations.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

E. Architectural Advantages

AIRE's modular design offers several key advantages:

- Separation of static extraction and semantic reasoning
- Scalability to large assemblies
- Compatibility with future updates of AI models
- Extensibility to other .NET languages
- Integration with automated cybersecurity pipelines

This layered structure allows AIRE to be flexible, transparent, and extendible to accommodate future research in the field of AI-based reverse engineering.

IV. Methodology

The methodology of AIRE is based on a structured multi-stage pipeline that operates on compiled assemblies written in the .NET version of the Visual Basic language. It follows a series of major stages that involve the loading of the assemblies, IL parsing, structured serialization, and the application of the AI engine. Each of these stages operates independently.The interaction workflow between the application components is shown in Fig. 2.

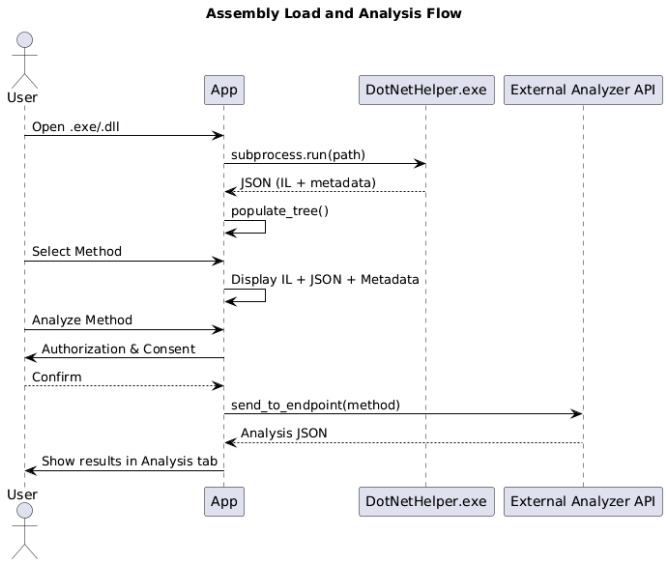

Fig. 2. Assembly loading and analysis workflow illustrating interactions between the user, application interface, .NET helper, and external AI analysis service.

A. Stage 1: Assembly Loading and Validation

The first stage involves the ingestion of the target VB.NET Windows x64 assembly, which is an executable file (`.exe`) or a dynamic link library (`.dll`). The system performs structural validation to verify that the binary is in the Common Language Infrastructure (CLI) format.

The following steps are performed during this stage:

- Verification of assembly format and metadata integrity
- Extraction of assembly-level descriptors (name, version, architecture)
- Enumeration of namespaces and type definitions
- Filtering of compiler-generated and system-level artifacts

It is loaded into memory in a read-only mode to prevent any changes from being made to it. This ensures that the analysis is a pure static analysis and does not involve any dynamic execution or instrumentation.

B. Stage 2: Intermediate Language (IL) Parsing

In the second phase of the AIRE approach, the analysis is performed at the method level with great precision. Each type present in the container is iterated over, and the analysis is performed at the method level. For each method, the system retrieves the following:

- Fully qualified method signature
- Access modifiers and attributes
- Parameter types and return type
- Local variable declarations
- Exception handling regions

The body of the method is then parsed on a per-instruction basis. For each instruction in the intermediate language (IL), the following information is tracked:

- Instruction offset
- Opcode mnemonic
- Operand type and value
- Control flow transitions
- Stack behavior implications

This phase maintains the exact sequence and structural integrity of IL code. Control flow constructs, including loops, conditional branches, and exception handlers, are mapped in a manner that is logically consistent. Unlike decompilation, there is no attempt to generate high-level source code. The goal is to preserve a representation at the IL level that is exact as well as analyzable.

C. Stage 3: Structured Serialization and Data Modeling

The extracted data is then converted into a hierarchical JSON schema, a data interchange format that can easily be interpreted by computers.

$$\text{Assembly} \rightarrow \text{Types} \rightarrow \text{Methods} \rightarrow \text{IL Instructions}$$

Each method object contains:

- Structural metadata
- Ordered instruction list
- Stack-related annotations
- Exception handling metadata

The serialization ensures that the resulting data is:

- Deterministic formatting
- Preserves the ordering of the instructions
- Consistent schema validation
- Remains compatible with AI processing engines

This form of data representation acts as an intermediate semantic layer between the low-level IL and the high-level reasoning models.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

D. Stage 4: AI-Based Semantic Inference

In the final stage, the structured JSON data is fed into the AI Semantic Analysis Engine for interpretation. The AI model applies contextual reasoning on the method-level IL patterns and derives high-level semantic understanding of the code. It determines the function of the code, predicts the input/output of the code, examines the stack manipulation patterns, and identifies suspicious API usage and vulnerabilities. Based on the patterns, the system creates human-readable semantic summaries. It is imperative to understand that the AI engine does not actually execute the code. Instead, the AI engine relies on the structural patterns and contextual information that exist within the IL instructions.

E. End-to-End Processing Flow

The overall flow of the process can be described as follows:

Binary Assembly $\rightarrow$ IL Extraction $\rightarrow$ JSON Structuring $\rightarrow$ AI Semantic Analysis $\rightarrow$ Structured Explanation

This flow is an end-to-end process that can be described as a multi-stage pipeline. This is an advantageous feature, especially in the context of large-scale assemblies. This is because combining static structural analysis with the AI engine is an effective solution for semantic reverse engineering.

V. Experimental Evaluation

In order to test the efficiency and performance of AIRE, experiments have been carried out on different VB.NET Window x64 assemblies with varying complexity in structure and size. The experiments have been carried out based on the following three main dimensions.

A. Experimental Setup

All experiments were carried out on a system with the following configuration:

- Processor: Intel Core i7 (8-core)
- RAM: 16 GB
- OS: Windows 11 (64-bit)
- .NET Runtime: 6.0

The test data set included:

- Small assemblies: 50–150 methods
- Medium assemblies: 300–800 methods
- Large assemblies: 1000+ methods

All code was compiled as follows: VB.NET, Windows x64 target architecture. Each assembly was analyzed separately to determine processing latency and memory consumption.

B. Performance Metrics

The following metrics were used to determine the performance of the system:

1) IL Extraction Time: The time taken to load, parse, and serialize the assembly data was measured. The results show that the extraction time varies according to the size of the assembly code. The results are as follows:

- Small assemblies: 0.4-0.8 seconds
- Medium assemblies: 0.9-1.5 seconds
- Large assemblies: 2.1-3.4 seconds

The extraction time was found to increase almost linearly with the number of methods.

2) Memory Overhead: The memory overhead of the IL code extraction process was found to remain below 300 MB for large assemblies. The memory requirement was found.

3) Metadata Extraction Accuracy: For the validation of structural integrity, the extracted metadata was cross-verified using manual IL inspection tools. The system was able to achieve:

- 100% coverage of metadata for methods
- Correct maintenance of order for individual instructions
- Correct mapping of exception handling blocks

No structural issues were found.

4) Semantic Inference Evaluation: For the evaluation of semantic reasoning using the AI system, a few methods were manually annotated. These annotations were used to determine the function of the methods. The summaries were cross-verified.

The system was able to:

- Correctly determine the function of the methods
- Correctly identify file I/O-related APIs
- Correctly identify network-related APIs

The semantic module was able to significantly reduce the manual effort required for interpretation.

C. Scalability Analysis

The overall complexity of extraction can be defined as follows, where n is the number of methods and m is the total IL instruction count:

$$O(n+m)$$

The reason this has linear time complexity with respect to the assembly code size is that each method and instruction is processed once.

D. Observations

This is further validated by the experimental results, which show that AIRE is capable of efficient and scalable IL extraction with structural fidelity.

Some of the key observations made are as follows:

- Linear scalability is achieved as the assembly size increases.
- Memory overhead is low compared to the instruction count.
- Structural accuracy is achieved for all test cases.
- Manual reverse engineering is reduced significantly.

This shows that the proposed method is successful in combining structured static analysis with AI semantic abstraction for reverse engineering purposes.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

VI. Semantic Reasoning Integration

One of the main contributions of AIRE is that it is able to transform low-level Intermediate Language (IL) representations into a semantic abstraction that is suitable for automated reasoning, vulnerability evaluation, and knowledge graph construction. This section will discuss the semantic transformation pipeline and how it helps bridge instruction-level disassembly and security interpretation at a higher level.

A. From IL Representation to Semantic Abstraction

IL instructions define stack-based operations and control flow constructs. The instructions are syntactically exact but do not have an explicit semantic definition of the functional intent. AIRE introduces the semantic abstraction level, which maps the structural patterns of IL code into higher-level behavioral representations.

If the method is defined by the ordered list of instructions as follows:

$$M = \{i_1, i_2, i_3, ..., i_k\}$$

i_j is defined by an opcode and an optional operand, the semantic transformation function can be defined as:

$$S(M) = f(\{i_1, i_2, ..., i_k\})$$

$S(M)$is the semantic summary, and f is the inference function driven by the AI.

B. Rule-Based Behavioral Mapping

AIRE applies predefined rules of behavior to identify known patterns of interest. These rules are based on structured JSON data and identify combinations of IL instructions related to particular behavioral categories.

Some examples include:

- **Suspicious API Call Detection:** File System Access, Network Communication, Registry Manipulation, and Dynamic Code Loading API calls.
- **Unsafe Stack Manipulation:** Irregular stack operation detection and excessive push-pop detection.
- **Exception Handling Anomalies:** Empty catch blocks and suppressed exception flows.
- **Reflection and Dynamic Invocation Patterns:** Potential detection for code obfuscation and runtime code generation.

The above rule-based systems ensure that deterministic security flags are generated before AI reasoning is applied.

C. AI-Driven Contextual Reasoning

In addition, the AI Semantic Engine also performs contextual reasoning, which is based on the method-level representation. Here, the AI model considers the following information for the purpose of contextual reasoning:

- Instruction sequences
- Control flow transitions
- API invocation patterns
- Data flow relationships

On the other hand, the output generated by the AI model for the purpose of contextual reasoning includes the following information:

- High-level function intent
- Risk classification (low, medium, high)
- Behavioral summaries
- Potential misuse scenarios

D. Knowledge Graph Construction

A JSON output with a structured format can be converted into a graph-based model with the following properties:

- Nodes can be methods, APIs, or data structures
- Edges can be invocation, dependency, or control flows

Formally, the program can be defined as a directed graph with the following properties:

$$G = (V, E)$$

where V is the set of program entities, and E is the relationship set based on the analysis of the IL.

This graph-based model allows for the following operations:

- Semantic similarity analysis
- Malware family clustering
- Policy compliance verification
- Automated reasoning via graph traversal

E. Explainability and Human-Readable Output

Another important goal of AIRE's design is explainability. Each semantic inference operation includes supporting IL evidence.

F. Security-Oriented Applications

The semantic reasoning layer supports several cybersecurity applications, including:

- Automated vulnerability triage
- Detection of insecure coding practices
- Malware behavior profiling
- Reverse engineering education support
- Automated technical documentation generation

AIRE bridges the gap between low-level disassembly and high-level security reasoning frameworks by integrating structured static analysis with AI-powered semantic interpretation.

VII. Limitations

While the AIRE approach shows promising results for AI-assisted semantic reverse engineering, some limitations need to be considered.

Firstly, the current approach is only applicable to VB.NET Windows x64 assemblies. It is true that the IL code is common across .NET languages; however, language-specific code and compiler optimizations may cause semantic differences, which the current approach does not address.

Secondly, the AI-assisted approach is based on structural patterns and interpretation; hence, the approach does not execute the code. As a result, the approach may not capture some

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

dynamic code behavior such as reflection, JIT compilation, and environmental code execution.

Thirdly, the approach may not perform well with obfuscated code using advanced techniques such as control flow flattening, string encryption, dynamic method calls, and instruction substitution. While the approach uses a rule-based approach for detection, the performance with advanced obfuscation is still a research challenge.

Fourth, semantic evaluation was performed using controlled data sets. Real-world large-scale malware benchmarking using precision and recall metrics is considered future work.

Lastly, the AI reasoning component is heavily dependent on the quality and alignment of the language model. Variability in the interpretation of the language model may cause minor inconsistencies in the overall summary.

However, the modular architecture of AIRE provides a pathway for improvement with regard to the above challenges.

VIII. Conclusion and Future Work

This paper presents AIRE, an artificial intelligence-based semantic framework for reverse engineering VB.NET Windows x64 assemblies. The framework utilizes artificial intelligence in the form of structured Intermediate Language (IL) extraction and semantic reasoning to improve the level of understanding in a Windows environment. The proposed framework is based on artificial intelligence and structured IL extraction to improve the level of understanding in a Windows environment. The proposed framework improves upon existing reverse engineering techniques that are mostly syntactic in nature and are based on source code reconstruction techniques.

The experimental results reveal that the proposed framework has linear scalability in terms of the number of methods and instructions with high structural fidelity during metadata extraction. The proposed framework reduces human interpretation through artificial intelligence-based semantic reasoning to improve the level of understanding in a Windows environment. The proposed framework bridges the gap between low-level disassembling and high-level reasoning to improve the level of understanding in a Windows environment. The proposed framework has been designed to be scalable in terms of artificial intelligence-based reverse engineering techniques and can be used in educational institutions to improve student knowledge in the domain of reverse engineering techniques.

A. Future Work

Although the current implementation shows encouraging results, the following research extensions are planned to improve the overall robustness and usability of the framework:

- **Cross Language Support:** Enhance the IL analysis capability for other .NET languages like C# and F#.
- **Graph Neural Network Support:** Map the programs into graph-based representations and leverage Graph Neural Networks (GNNs) for improved prediction accuracy and malware family classification.
- **Real-Time Malware Scanner:** Design an automated framework capable of real-time assembly code inspection and semantic risk scoring for deployment in the enterprise environment.
- **Resisting Obfuscation:** Improve the overall framework robustness against control flow flattening, instruction subversions, and reflection-based malware obfuscations.
- **Quantitative Semantic Evaluation:** Include quantitative measurements like precision, recall, and F1-score for the accuracy of the semantic inference results against manually annotated datasets.
- **Knowledge Graph Support:** Facilitate the creation of large-scale knowledge graphs for binary programs for detecting similarities and ensuring compliance.

The future directions of this research aim to integrate static structural analysis with sophisticated machine learning models to further automate semantic reasoning within compiled code environments. AIRE's advances in explainable AI-assisted reverse engineering contribute to the overall goal of developing intelligent cybersecurity analysis systems that can understand complex binary code with little to no human intervention.

Acknowledgment

The authors wish to extend their sincerest gratitude towards the Department of Networking and Communications, SRM Institute of Science and Technology, for providing the necessary support for carrying out the research.

The authors of the paper wish to thank the mentors and reviewers for providing valuable feedback and guidance while developing the AIRE framework, which significantly contributed towards the development of the paper.

References

[1] C. Eagle, *The IDA Pro Book*, 2nd ed., No Starch Press, 2011.
[2] ECMA International, "Common Language Infrastructure (CLI)," ECMA-335 Standard, 2012.
[3] D. Ucci, L. Aniello, and R. Baldoni, "Survey of machine learning techniques for malware analysis," *Computers & Security*, vol. 81, pp. 123–147, 2019.
[4] S. H. Ding, B. C. M. Fung, and P. Charland, "Asm2Vec: Boosting static representation robustness for binary clone search," in *Proc. IEEE Symposium on Security and Privacy*, 2019.
[5] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis," in *Proc. ACM Symposium on Principles of Programming Languages*, 1977, pp. 238–252.
[6] M. Christodorescu and S. Jha, "Static analysis of executables to detect malicious patterns," *IEEE Security & Privacy*, vol. 3, no. 6, pp. 66–73, 2005.
[7] B. Schwarz, D. Grunwald, and J. Schneier, "Static detection of malicious code in binaries," in *Proc. Annual Computer Security Applications Conference*, 2001.
[8] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Proc. IEEE Symposium on Security and Privacy*, 2014.
[9] Microsoft Corporation, ".NET Runtime Documentation," Microsoft Docs, 2023.
[10] N. Viennot, E. Garcia, and J. Nieh, "A measurement study of Google Play," in *Proc. ACM SIGMETRICS*, 2014.
[11] X. Zhou and R. State, "Feature extraction for malware detection in .NET binaries," *IEEE Access*, vol. 8, pp. 152147–152160, 2020.