




Detecting Malicious .NET Executables Using Extracted Methods Names

Hamdan Thabit¹, Rami Ahmad¹ , Ahmad Abdullah¹, Abedallah Zaid Abualkishik¹  and Ali A. Alwan^{2,*} 

¹ College of Computer Information Technology, American University in the Emirates, Dubai 503000, United Arab Emirates; 191120019@aue.ae (H.T.); rami.alshwaiyat@aue.ae (R.A.); 191120017@aue.ae (A.A.); abedallah.abualkishik@aue.ae (A.Z.A.)

² School of Theoretical & Applied Science, Ramapo College of New Jersey, Mahwah, NJ 07430, USA

* Correspondence: aaljuboo@ramapo.edu

Abstract: The .NET framework is widely used for software development, making it a target for a significant number of malware attacks by developing malicious executables. Previous studies on malware detection often relied on developing generic detection methods for Windows malware that were not tailored to the unique characteristics of .NET executables. As a result, there remains a significant knowledge gap regarding the development of effective detection methods tailored to .NET malware. This work introduces a novel framework for detecting malicious .NET executables using statically extracted method names. To address the lack of datasets focused exclusively on .NET malware, a new dataset consisting of both malicious and benign .NET executable features was created. Our approach involves decompiling .NET executables, parsing the resulting code, and extracting standard .NET method names. Subsequently, feature selection techniques were applied to filter out less relevant method names. The performance of six machine learning models—XGBoost, random forest, K-nearest neighbor (KNN), support vector machine (SVM), logistic regression, and naïve Bayes—was compared. The results indicate that XGBoost outperforms the other models, achieving an accuracy of 96.16% and an F1-score of 96.15%. The experimental results show that standard .NET method names are reliable features for detecting .NET malware.

Keywords: malware analysis; malware detection; windows; static analysis; .NET; machine learning



Academic Editor: Gianni D'Angelo

Received: 6 August 2024

Revised: 5 January 2025

Accepted: 17 January 2025

Published: 21 January 2025

Citation: Thabit, H.; Ahmad, R.; Abdullah, A.; Abualkishik, A.Z.; Alwan, A.A. Detecting Malicious .NET Executables Using Extracted Methods Names. *AI* **2025**, *6*, 20. <https://doi.org/10.3390/ai6020020>

Copyright: © 2025 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

Since its establishment in 2002, the Microsoft .NET framework has become essential for cross-platform application development and is known for its flexibility and comprehensive ecosystem that includes a variety of programming languages, including C#, VB.NET, and F# [1]. Features such as just-in-time (JIT) compilation and extensible class libraries equip .NET to create scalable and secure software. However, these same features can also simplify the process of developing malicious Windows malware by attackers. Statistical studies also reveal a significant increase in malware targeting Windows devices over the past decade, highlighting the growing interest in security measures [2].

In Windows malware analysis, the difference between static analysis and dynamic analysis has become particularly obvious [3]. Static malware detection scans the content of Portable Executable (PE) files without actually executing the malware samples. During static analysis, analysts extract features, including string patterns, opcodes, and byte sequences [4]. The extracted features are used to determine whether a file is malware or

not. However, static analysis may quickly identify characteristics of known malware [5], although it struggles against unknown and zero-day threats as well due to its reliance on signature databases [6,7]. Next, dynamic analysis of malware behavior includes extracting features such as network behavior, registry changes, system calls, and memory usage. Windows Application Programming Interface (API) call sequences are one of the most representative features in behavior-based malware detection [8]. Dynamic analysis [4] provides crucial insights into malware behavior during execution but is resource-intensive and also poses risks for sandbox escapes.

The .NET Framework consists of two primary components: the Common Language Runtime (CLR) and the .NET class library. The CLR acts as a critical layer between the Common Intermediate Language (CIL) code and native machine instructions [9], enabling language-agnostic execution and platform independence. Unlike C/C++ applications, which are compiled directly into machine code, every .NET application—whether written in C#, VB.NET, or any other .NET-compatible language—is compiled to CIL and stored in assembly formats such as .dll or .exe. Upon execution, the CLR compiles these instructions into machine code at runtime [10].

Consequently, we must be aware that current malware detection methods use APIs based on static APIs or dynamic API sequences to classify malware [11] and are primarily designed to target malware executables written in C/C++. The static APIs extraction method using existing PE analysis tools encounters limitations when applied to malware written in .NET languages. Unlike C/C++ binaries, where APIs can be statically extracted from the import table, this approach is not directly applicable to .NET binaries due to differences in their Portable Executable (PE) structures [12]. In .NET, method references are stored in metadata and Intermediate Language (IL) code, which requires specialized tools to parse and analyze, as these methods are not listed in the import table in the same way as in native binaries.

In .NET environments, the Common Intermediate Language (CIL) is generated during compilation instead of machine code [13]. Moreover, the reliance on dynamic API sequences collected through executing malware in a sandbox makes the classification process slower [7].

Recent studies on malware detection in Windows have mostly focused on analyzing API sequences rather than investigating .NET-specific features, as shown in research by Zhang et al. [14]. The authors in [14] used dynamic analysis of malware with semantic analysis of API sequences, which, although reliable, still brings notable security risks linked with malicious code execution. Furthermore, “API-MalDetect” [2] used deep learning to distinguish between benign and malicious API patterns, presenting the possibility of an accurate assessment capable of identifying complex malware activities hidden within real processes. However, these machine learning methods rely heavily on the availability of large, unique, and well-organized datasets that contain API call sequences extracted from executing the samples in an isolated sandbox. This requires large computational resources, which makes their implementation difficult in resource-constrained environments. Similarly, many studies, such as [8,15–20], have used deep learning techniques to analyze malware through applications of API call patterns. These investigations have mostly focused on general environments without specific consideration of the .NET framework. Thus, this oversight highlights a major gap, as none of these studies explicitly discuss or utilize the unique attributes of the .NET environment in malware detection methodologies.

Recognizing the challenges posed by the .NET environment as we discussed earlier, which is highly affected by malware attacks, this study presents a new framework specifically designed for .NET applications. We aim to reduce the risks associated with executing potentially malicious code by leveraging advanced decompilers like dnlib [21] to statically

extract standard .NET method names from .NET executables. Our methodology also uses machine learning models to analyze the extracted methods, which will be trained on a structured dataset containing benign and malicious .NET executables.

The main contributions of this paper can be summarized as follows:

- Develop a framework for detecting malicious .NET executables using extracted method names.
- Create a dataset by collecting malware and benign .NET executables from online sources and extract .NET methods from these samples.
- Evaluate the effectiveness of accurately detecting .NET malware using only .NET method names.
- Compare the performance of different machine learning models and identify the most accurate model for .NET malware detection.
- Evaluate the impact of feature length on the classification accuracy of different machine learning models, including XGBoost [22], random forest [23], K-nearest neighbor (KNN) [24], support vector machine (SVM) [25], logistic regression [26], and naïve Bayes [26] in .NET malware detection.

The organization of this paper is as follows: Section 2 reviews related works, identifying gaps and setting the stage for our contributions. Section 3, “Preliminary”, introduces essential concepts and definitions. In Section 4, we present our proposed framework. Section 5 examines the empirical evaluation of the model’s performance, which includes both results analysis and discussion (Results and Discussion). In Section 6, we discuss the limitations and future work of our research. The paper concludes in Section 7, summarizing key findings.

2. Related Work

Malware detection within Windows operating systems is becoming more sophisticated, driven by continued advances in malware techniques that exploit the complexities of modern computing environments [6]. As we mentioned in the introduction, most of the recent studies that work on malware detection in Windows are mostly based on API call sequences. Furthermore, machine learning and deep learning models have been relied upon to improve malware detection accuracy in both static and dynamic techniques. Refs. [2,14] demonstrated the use of deep learning to analyze semantic differences in API call sequences, providing a more precise understanding of malware behavior. Although these techniques are promising, they require large, constantly updated training datasets. Another work, ref. [11], proposed combining static and dynamic analysis with deep learning to overcome the limitations of traditional techniques. The model improves detection by considering not only the frequency and sequence of API calls but also the arguments, which traditional detectors often ignore. Furthermore, deep learning techniques were applied to identify important features and capture contributions from both types of analysis, ensuring a balanced discovery approach.

Furthermore, a long short-term memory (LSTM) network approach has been used to classify malware based on API call sequences [20]. LSTM models are able to learn complex patterns over time, making them well suited for identifying malware that displays modifications in behavior during its execution. However, the computational overhead of training these models as well as the demand for large datasets for training present significant difficulties that may delay their functional deployment. Another related effort [19] used a sequence of recurrent neural networks (RNNs) and LSTM to dynamically parse API call sequences, aiming to overcome the limitations imposed by code obfuscation in static analysis and evasion techniques in dynamic analysis. The work used a series of steps that include data preprocessing, n-gram-based feature extraction, and inverse frequency

document frequency for feature selection and orientation in training the RNN-LSTM model. However, this approach achieved an impressive 92% accuracy rate based on a dataset of 3000 different malware and benign traces. Challenges such as the computational overload of training and the need for large, representative datasets remain significant obstacles that may hinder wider distribution.

Transfer learning has been explored by various studies to improve malware detection performance. One such study [27] leverages transfer learning by fine-tuning the pre-trained language model GPT-2 on API call sequences. The primary objective is to develop a model capable of predicting and identifying malicious behavior patterns in the early stages of Windows malware execution. Fine-tuning allows the model to adapt to the specific domain of API call sequences, improving its ability to recognize patterns of malicious behavior that are otherwise difficult to detect using traditional methods. Another study [28] used transfer learning by fine-tuning the pre-trained InceptionV3 model on malware signature representations formatted as 2D images. The resulting approach yielded a good performance, surpassing that of comparative models such as LSTM.

In different techniques, the authors in [4] used a combination of word embedding and Markov chain modeling to analyze API call sequences. They start by using word embedding to group API functions based on contextual similarities and then cluster these functions to simplify API sequences. A semantic chain transition matrix is then developed to capture the relationships between the clusters, which helps in constructing a Markov chain model. This approach has demonstrated high accuracy using large-scale datasets; however, this technique requires significant computational resources for training and may not scale efficiently in resource-constrained environments, representing a limitation in its applicability. An additional innovative method for in-memory malware analysis has been developed for .NET applications [29]. This method focuses on identifying anomalies in how functions allocate and manage memory, which may indicate the presence of malicious activity. Although this approach provides a direct way of detecting malware by examining its behavioral patterns in system resource use, it requires extensive system access and may not be suitable for environments with strict privacy or operational constraints.

To address these challenges, the current research proposes an entirely new framework leveraging static method extraction techniques tailored specifically for the .NET framework. This framework aims to bypass the difficulties and resource-intensity issues associated with dynamic analysis of executing suspicious code. By using machine learning algorithms to evaluate extracted .NET methods, this work aims to create a detection system that is less demanding on operational resources while still being reliable.

3. Preliminaries

This section explains the basic principles of the .NET malware detection framework and discusses static and dynamic analysis techniques and their unique advantages and challenges. It highlights the differences between .NET and C/C++ executables, focusing on the essential aspects of efficient .NET methods extraction. Additionally, it explains the process of decompiling .NET executables.

3.1. Malware Analysis Techniques

There are two basic methods for malware analysis: static analysis and dynamic analysis. Each method offers unique advantages and disadvantages when it comes to detecting and countering malware.

A. Static analysis

Static analysis involves examining malware without executing its code. This method is useful in identifying malware attributes, including metadata, strings, structure, and

code, making it a cornerstone of signature-based detection systems [30]. This technique is effective in detecting known malware and is used by signature-based detection systems by comparing the provided code signature with a database containing known malware signatures. On the other hand, static analysis faces many challenges in effectively identifying and analyzing malware. Malware developers often use obfuscation and encryption to hide the malicious functionality of their code, making it difficult for static analysis tools to detect the true functionality [31].

B. Dynamic analysis

Dynamic analysis refers to a technique used in malware analysis where the behavior of suspicious code or software is examined by observing its actions in a secure, controlled environment [14]. This method offers several advantages, such as the ability to extract obfuscated APIs from encrypted or obfuscated malware by observing system calls [4]. It also detects evasive malware that modifies behavior when detected or remains dormant and provides complete visibility of malware behaviors like network communication, file manipulation, and registry changes. However, dynamic analysis has several disadvantages. Its analysis time is slower than static analysis due to the need to create a secure environment, execute the malware, and monitor its activities. It also places considerable demands on computational resources [30]. Moreover, there is always a risk that the malware may escape the controlled environment, potentially infecting other systems or networks.

3.2. .NET vs. C/C++ Executables

Knowing the differences between the execution and combination of .NET and C/C++ executables is essential for malware evaluation, especially when using .NET method extraction techniques.

A. .NET Executables

.NET Framework architecture is illustrated in Figure 1. .NET executable files (.NET applications) are converted into CIL, also recognized as bytecode, during the compilation process. IL instructions, unlike machine code, are not executed directly by the hardware but rather by the CLR [32]. The CLR utilizes a technique called JIT compilation to transform CIL into machine code at runtime. Thus, .NET executables exhibit platform independence until JIT-compiled, as the same .NET application can function on any system that supports the .NET framework or .NET Core. This flexibility is particularly advantageous in multi-platform environments [33].

B. C/C++ Executables

C/C++ executable files are compiled directly into machine-specific instructions intended for the target architecture, resulting in code that can be executed directly by the hardware without the need for an intermediary. Due to this, C/C++ executables possess less portability than .NET-CIL, as a C/C++ program compiled for an x86 architecture will not run on Advanced RISC Machine (ARM) [34] without recompilation for the ARM platform. The C/C++ executables that are compiled into machine code for a specific target architecture expose a reduced degree of system independence, while .NET-CIL is more adaptable across various platforms due to its abstract and non-machine dependent.

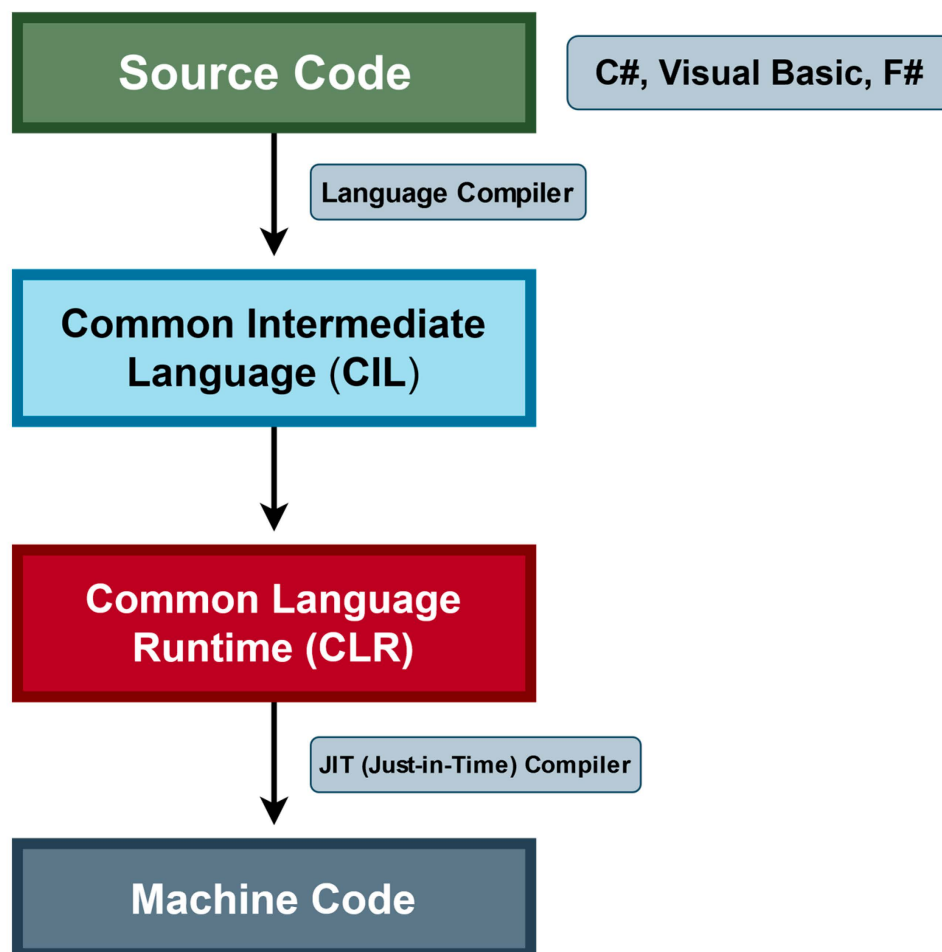


Figure 1. .NET framework.

3.3. Decompiling .NET Executables for Malware Analysis

The main feature of our dataset is the .NET method names. We can extract method names from samples dynamically or statically. Dynamically extracting .NET method names involves executing malware in a sandbox environment. This approach allows us to trace the executed method names in their actual order, resulting in an accurate representation of its method usage. However, it comes with significant drawbacks. The process is time-consuming due to the need for creating and managing a secure sandbox, and it requires substantial hardware resources. Instead, we used the static extraction method for more efficiency. To achieve this, we used the dnlib library [21] to decompile the .NET executables. After the decompilation, we parse the decompiled code and systematically extract the .NET methods. This technique is faster and more resource efficient than dynamically extracting .NET methods from malware execution.

4. Proposed Framework

In this section, we present a framework specifically developed for using .NET extraction methods for advanced Windows malware detection. Our methodology involves several important stages, including collecting .NET samples, extracting .NET methods, creating the dataset, and applying machine learning models. Details of this framework are shown in Figure 2.

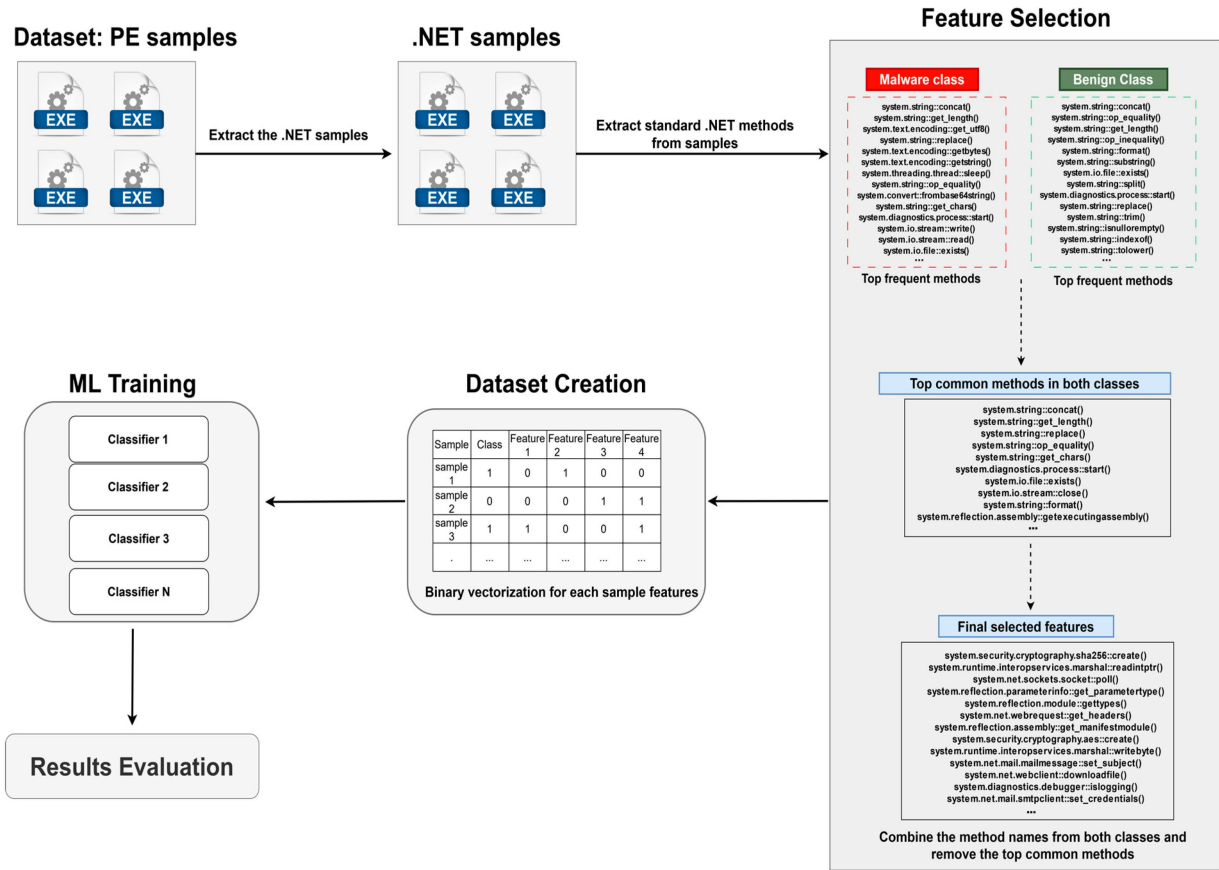


Figure 2. The proposed .NET malware detection framework.

4.1. Portable Executable Samples Collection

To address the lack of publicly available datasets focused exclusively on .NET samples, we built a new dataset that contains only .NET samples for the purpose of this research. The total number of collected samples was 148,645 samples. All the samples were Windows executables, which could be written in various programming languages such as C++, Go, or Rust. Malware samples were obtained from MalwareBazaar [35] and VirusShare [36], while benign samples were downloaded manually from SourceForge [37] and Github [38].

To extract .NET samples from the collected samples, we searched for the presence of the “IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR” directory in the Portable Executable (PE) optional header. This directory points to the CLR (Common Language Runtime) header. If present, then the PE file is a .NET executable. We found this method is the most reliable method to determine if a PE executable is .NET executable. We automated this process using Python and the pefile library [39], which enables us to read PE headers efficiently. Out of 143,397 malware PE samples we collected, we extracted 8759 .NET samples, representing approximately 6% of all collected malware PE samples.

4.2. Features Extraction: .NET Methods Extraction

Feature extraction is an important step in transforming our dataset into a format suitable for machine learning models. In this process, we aimed to identify and extract standard .NET methods from decompiled .NET executables. We focused on extracting standard .NET methods that can be imported from standard .NET libraries such as System.IO and Microsoft.Win32. A standard .NET library is a collection of pre-built classes, interfaces, and functions that are bundled together as part of the .NET Framework. The parameter values passed to the method were ignored since they do not hold significance

for our analysis as we focused on detecting the presence of specific methods within a sample. Instead, we aimed at determining if a particular method was present or absent in the code, regardless of its usage context. Each sample should contain a minimum of 10 different method names to be included in the final dataset. At least 10 different method names were selected to ensure proper representation of features, reduce noise, enhance model robustness, improve classification accuracy, maintain balance of the dataset, and simplify the feature extraction process, thus building a high-quality dataset for effective machine learning exercise. Algorithm 1 explains the steps of the feature extraction process and building a high-quality dataset for reliable machine learning training.

Algorithm 1: .NET Features extraction

Input: Directory path of .NET executables samples

Output: Features of each sample written in a text file

Begin

source_dir ← Directory path of .NET executables samples

result_dir ← Directory path of the samples features

for each file in source_dir **do**

 load the assembly file and iterate through modules, types, and methods

for each method **do**

if method is a .NET standard method **then**

 preprocess the method name

 add the cleaned method name to a HashSet of .NET method names

end if

end for

if the total number of unique .NET method names is ≥ 10 **then**

 create a new text file in result_dir

 write methods names from the HashSet to a text file

else: skip to next assembly

end if

end for

In order to analyze .NET executables, we first decompile them using dnlib [21]. The decompiled sample consists of multiple interconnected modules. Each module represents a separate unit within the executable that can be analyzed independently.

Subsequently, we iterate over each module in the decompiled sample with the goal of extracting all the methods it contains. For every method, we check if the method is a standard .NET method by checking the namespace of the method. If a method is a standard .NET method, we preprocess the method name by removing the passed arguments to it and convert the method name to lowercase before adding its preprocessed name to a hashset of extracted features for the sample. The total unique methods of each sample must equal or be larger than 10 to be added to the dataset. Next, we write the preprocessed method names to text files. Each text file is labeled with the SHA256 hash value of its corresponding sample to maintain a well-organized dataset for further analysis. Samples with fewer than 10 unique method names are ignored as they may not yield sufficient context for accurate classification. Figure 3 shows some features extracted from the files.


```
System.Reflection.MethodInfo::get_ReturnType()  
System.Reflection.ParameterInfo::get_ParameterType()  
System.Reflection.Module::ResolveMethod()  
System.Reflection.Emit.DynamicILInfo::SetLocalSignature()  
System.Reflection.Emit.DynamicMethod::CreateDelegate()  
System.Reflection.Emit.DynamicILInfo::SetCode()  
System.Reflection.FieldInfo::get_FieldType()  
System.Reflection.FieldInfo::SetValue()  
System.Reflection.MemberInfo::get_DeclaringType()  
System.Reflection.Emit.DynamicMethod::GetDynamicILInfo()  
System.Convert::ToInt32()  
System.Reflection.FieldInfo::GetFieldFromHandle()  
System.Reflection.Module::ResolveSignature()  
System.Reflection.MethodBase::get_IsStatic()  
System.Reflection.MethodBase::get_IsConstructor()  
System.Reflection.Emit.DynamicILInfo::GetTokenFor()  
System.Reflection.MemberInfo::get_MetadataToken()  
System.Reflection.MemberInfo::get_Module()  
System.Reflection.MemberInfo::GetCustomAttributes()  
System.Reflection.FieldInfo::GetOptionalCustomModifiers()  
System.Reflection.MethodBase::GetParameters()  
System.Reflection.MemberInfo::get_Name()  
System.Reflection.MethodBase::get_MethodHandle()  
System.String::get_Chars()
```

Figure 3. Sample of .NET extracted features.

4.3. Dataset Creation

Creating our dataset involves selecting method names based on their frequencies of occurrence within both malware and benign samples. A limit of 50 occurrences (frequency threshold) is set for each method name frequency in each class to filter out rare method calls. Moreover, this threshold reduces the overall dimension of the dataset by reducing the number of selected method names from each class. Increasing the frequency threshold value results in a lower number of final features, as less common method names are excluded. Choosing to use a threshold of 50 is a strategic decision aimed at optimizing the dataset for subsequent analysis. It is designed to ensure that the data used to train machine learning models is manageable and meaningful, increasing the possibility of accurate classification while reducing noise and irrelevant information.

The steps of feature selection process are as follow:

1. Count the frequency of method names within both the malware class and the benign class.
2. Filter out method names with a frequency below a predefined threshold (frequency threshold equals to 50 in our case) from both classes.
3. Identify the top 30 most frequent method names in each class separately.
4. Determine the set of common method names between malware and benign classes by intersecting the two sets.
5. Merge the feature set of both classes using the OR operation between the sets.
6. Remove any common method names from the merged feature set.

The feature selection process was conducted on the entire dataset prior to any under-sampling and splitting for training and testing. The use of the entire dataset for feature selection is relevant in malware detection research [40–42], where the goal is to identify

global features and patterns that are most discriminative across classes. Although separating feature selection from the training and testing datasets is theoretically ideal, it may not always be feasible, especially when the dataset is limited in size. In our case, we applied feature selection to the entire dataset since our feature selection method, which is based on frequency thresholding, requires sufficient data to accurately identify global features and patterns.

Our feature selection process uses label information. While labels were used during the feature selection step, this was a one-time preprocessing step. The final global features were fixed after this stage. During training and inference, the model relies only on the presence or absence of these features, ensuring no label dependence at inference time. The reason we chose to apply label-dependent frequency thresholding is to ensure that we explicitly remove top common features that are shared between both malware and benign samples. This process improves the feature selection, ensuring that the final set is not only statistically relevant but also discriminative. By separating the dataset into malware and benign samples, we can explicitly target the overlap of features between the two, which would be difficult to achieve in a purely label-independent process. However, it is important to note that the resulting set of 556 global features remained fixed and independent of the subsequent training and evaluation processes. This ensures that no iterative feedback influenced the classifier.

We then selected the top 30 method names from each category (malware and benign) to highlight the common features in each class. Choosing the top 30 method names instead of a larger number (e.g., top 100) allows us to find and remove the most common features while minimizing the removal of less common methods that may be essential for accurately representing the samples' features. Intersection analysis was applied to reduce the dimensionality of the dataset by finding and removing common features between malware and benign classes.

Figure 4 presents a Venn diagram that visually represents the top 30 most frequent methods within the malware class (red circle) and the top 30 most frequent methods in the benign class (green circle). The intersection of these two circles contains 16 methods that exist within both groups.

Table 1 also shows the frequency distribution of the top 10 most common methods (the intersection) between the malware and benign categories. Furthermore, the top 10 methods with the highest frequency under both malware and benign categories after removing common methods are shown in Table 2. Top frequently used methods in the malware class, as shown in Table 2, do not necessarily imply that they are not used by benign samples; instead, they are used less frequently compared to their usage within the malware class.

Table 1. Frequency of the common methods in malware and benign classes.

Method Name	Frequency in Malware Class	Frequency in Benign Class
system.string::concat()	1377	2864
system.string::get_length()	1091	1984
system.string::replace()	923	1325
system.diagnostics.process::start()	757	1368
system.string::op_equality()	810	2196
system.threading.thread::sleep()	795	801
system.string::get_chars()	841	920
system.io.stream::close()	705	944
system.io.file::exists()	727	1571
system.string::format()	735	1524

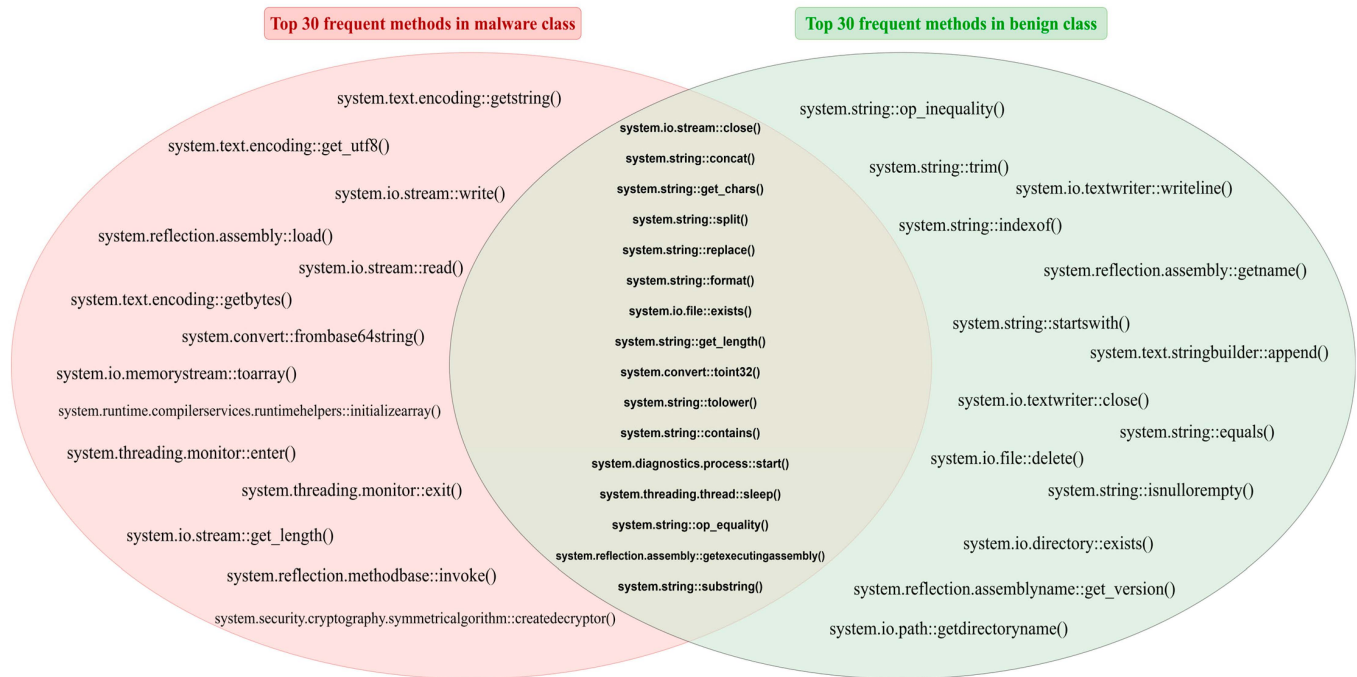


Figure 4. Venn diagram.

Table 2. Comparison of top 10 frequent method names in malware and benign categories.

Rank	Malware Class—Method Name	Count	Benign Class—Method Name	Count
1	system.text.encoding::getstring()	860	system.string::op_inequality()	1705
2	system.text.encoding::get_utf8()	855	system.string::trim()	1135
3	system.text.encoding::getBytes()	850	system.string::indexof()	1077
4	system.convert::frombase64string()	807	system.io.textwriter::writeline()	981
5	system.io.stream::read()	724	system.io.directory::exists()	927
6	system.io.stream::write()	720	system.string::startswith()	923
7	system.runtime.compilerservices.runtimehelpers::initializearray()	704	system.reflection.assembly::getname()	887
8	system.io.memorystream::ToArray()	674	system.text.stringbuilder::append()	872
9	system.reflection.assembly::load()	617	system.io.textwriter::close()	836
10	system.io.stream::get_length()	616	system.string::equals()	796

The final dataset consists of 556 unique .NET method names as features alongside two additional columns: “file” and “class”. The “file” column holds a unique hash value for each sample, while the “class” column represents the binary representation (0 for benign, 1 for malware) of each analyzed sample’s class label. The binary vectorization process is then applied to each feature name within our final dataset. This approach involves setting a value of either 1 or 0 based on the presence or absence of that specific method name within a given .NET sample’s method names. In our work, we used static analysis to extract method names (features) from decompiled code. The order of these method names cannot be known without executing the sample and intercepting the called methods. Binary vectorization is a suitable choice when the order of features is not important or known. However, in certain scenarios, like API call sequences, which are used in [2] and [20], the order of features is known, and it is important to preserve the order of the tokens during tokenization to provide context and ensure accurate classification by the ML classifier. An example of an appropriate tokenization method that considers the order of the tokens is word-level tokenization, which assigns a unique numerical value to each token in the vocabulary while maintaining their original sequence [43]. Lastly, samples containing

fewer than 10 features are excluded from the dataset due to insufficient context for accurate classification, as mentioned previously.

We used random sampling as a data balancing technique. As shown in Table 3, following the dataset creation process, the benign class comprised the majority of instances with 2435 examples, while the malware class consisted of 1598 instances. The reduction in the dataset creation process happens because there are samples that contain less than 10 features from the final selected features. Those samples are ignored because they do not provide enough context for ML classification. For a fair evaluation, we balanced both classes by randomly selecting an equal number of samples from each category, resulting in 1500 samples per class. Table 4 shows the number of samples for before and after the data balancing process with the reduction percentage.

Table 3. Number of samples before and after applying the dataset creation process.

Class	Before Dataset Creation Process	After Dataset Creation Process	Reduction Percentage
Malware	8759	1598	81%
Benign	5248	2435	37%

Table 4. Number of samples before and after applying the data balancing process.

Class	Before Dataset Balancing	After Dataset Balancing	Reduction Percentage
Malware	1598	1500	6%
Benign	2435	1500	38%

A. Machine Learning Training

In this work, machine learning plays a crucial role by automating the identification of malicious patterns and behaviors in .NET executables. Training models on a dataset derived from files with .NET method names (both benign and Malware) allows for accurate classification between malware and legitimate software. Six machine learning algorithms were selected based on their diverse strengths and applicability to binary classification tasks in malware detection. We chose to use XGBoost, random forest, KNN (K-nearest neighbors), SVM (support vector machine), logistic regression, and naïve Bayes. The XGBoost algorithm was selected due to its ability to handle high-dimensional data and capture complex patterns. Its regularization techniques make it robust against overfitting, particularly when working with noisy datasets. We considered the random forest because of its ensemble nature, which combines multiple decision trees to improve predictive accuracy. It is known for its resilience against overfitting and its ability to handle both categorical and continuous data, making it effective in capturing the variance in our dataset. The KNN is a non-parametric algorithm that performs well in detecting local data structures, which can be important when distinguishing between similar benign and malicious executables. However, it can be sensitive to the scale of the data and is less robust to noise, which is mitigated by the other models. The SVM was chosen due to its strength in handling binary classification problems. By maximizing the margin between classes, SVM is particularly effective when there is a clear boundary between malware and benign samples, though it can be computationally intensive on larger datasets. We selected logistic regression for its simplicity and interpretability. It serves as a strong baseline model, providing insight into how features contribute to the classification of malware. However, it may struggle with more complex patterns that are better captured by ensemble models. Lastly, Naïve Bayes

was included due to its effectiveness in high-dimensional spaces, like the one created by numerous method names in our dataset.

These models were chosen to complement one another, with each model bringing a unique set of strengths to mitigate potential weaknesses in the others. By employing this diverse set of algorithms, we ensured a more comprehensive evaluation of their effectiveness in detecting .NET malware.

5. Evaluation and Results

In evaluation and testing, the model was configured with Python 3.11 running on Debian 11 (64-bit) Linux, powered by a 2.90 GHz Intel(R) Core(TM) i5-9400 CPU with 32 GB of RAM. Moreover, the model's performance was evaluated using accuracy, precision, recall, as well as F1-score metrics. The formulas to determine these metrics are presented below:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \quad (1)$$

$$Precision = \frac{TP}{TP + FP} \quad (2)$$

$$Recall = \frac{TP}{TP + FN} \quad (3)$$

$$F1 = \frac{2 \times Precision \times Recall}{Precision + Recall} \quad (4)$$

The symbol TP stands for “true positives”, pertaining to cases where the model correctly predicts the positive class. The symbol TN signifies “true negatives”, which are scenarios through which the model correctly recognizes the negative category. FP represents “false positives”, where the model incorrectly forecasts the positive class, and FN means “false negatives”, where the model fails to correctly determine the positive class.

5.1. Experimental Results

Our experiment involved testing six different machine learning models—XGBoost, KNN, random forest, logistic regression, naïve Bayes, and SVM—on the built dataset consisting of 1500 malware samples and 1500 benign samples. The features length for all the samples is 556 features. We selected these models based on their popularity and effectiveness in classification tasks. The diverse selection of models would provide a comprehensive evaluation of their capabilities in our .NET malware detection task.

To ensure robustness and generalization, we evaluated the models using five-fold cross-validation. This technique involves splitting the dataset into five equal-sized subsets and training the model on four of them while testing it on the remaining subset. We repeated this process for each subset, averaging the results to obtain a final performance score. By averaging the evaluation metrics over all 5 folds, we minimized the risk of bias that may result from using a single train-test split, ensuring a more robust and reliable comparison between the models. To achieve an optimal balance between precision and recall in our classification task. We prioritized the F1-score as the primary criterion for model selection. The model with the highest average F1-score across the folds was selected as the best-performing model.

In terms of overall performance in the five-fold cross-validation, the XGboost model performed the best with an accuracy of 96.16%, followed by random forest with an accuracy of 95.36%. From the result shown in Table 5, we can see that all models performed well, with XGBoost having the highest accuracy and F1-scores among them. This shows that using XGBoost for this binary classification is the optimal choice. The superior performance achieved by XGboost is primarily due to its advanced ensemble learning approach, which

builds decision trees sequentially to correct the errors made by previous trees. This allows XGBoost to capture complex and non-linear patterns in the data effectively. Malware detection often involves complex feature relationships, making it difficult for models to achieve accurate results. However, XGBoost's gradient boosting framework is particularly effective at identifying these through its iterative process. Furthermore, XGBoost's ability to handle both large feature spaces contributed significantly to its performance, allowing it to distinguish between malware and benign samples with high precision and recall.

Table 5. Performance metrics of all six different models (5-fold cross-validation).

Model	Accuracy	Precision	Recall	F1-Score
XGBoost	96.16%	96.17%	96.14%	96.15%
Random forest	95.36%	96.94%	93.96%	95.28%
KNN	90.73%	90.25%	90.51%	90.71%
SVM	95.16%	96.41%	93.79%	95.08%
Logistic regression	95.3%	95.6%	94.92%	95.27%
Naïve Bayes	88.66%	91.79%	84.96%	88.24%

Random Forest demonstrated a strong performance with high precision (96.94%), but its recall (93.96%) was lower. This observation suggests that the model tends to favor precision over recall, potentially reducing false positives at the expense of missing some positive instances. SVM shows a similar pattern, with precision (96.41%) surpassing recall (93.79%). The model's reliance on optimizing the decision boundary through a margin-based approach likely explains its high precision but a slightly lower recall. Logistic Regression shows a strong performance with an accuracy of 95.3% and an F1-score of 95.27%. This model's strength lies in its simplicity and effectiveness in linearly separable data, although it lacks the flexibility of more complex models like XGBoost in capturing non-linear patterns.

KNN performed moderately, with an accuracy of 90.73% and an F1-score of 90.71%. The limitations of this model in handling complex decision boundaries can be attributed to its reliance on proximity-based decision making, which becomes less efficient in high-dimensional data as the number of features increases. Naïve Bayes shows the weakest performance across all models, with an accuracy of 88.66%. Naïve Bayes' assumption of feature independence may limit its effectiveness on datasets where feature interactions are important.

In Figure 5, the receiver operating characteristic (ROC) curve for each model is presented. Results indicated that all models achieved high area under the curve (AUC) scores. Random forest has a slightly better overall performance with an AUC score of 0.992 compared to XGBoost, which achieved an AUC score of 0.990. A higher AUC score indicates better capability to separate between the positive and negative classes, meaning that the model is more likely to correctly classify samples as either positive or negative.

The analysis of the confusion matrices for the six models highlights distinct differences in their ability to classify malware and benign samples. Figure 6 contains the confusion matrix on the test dataset for all the models. The KNN model demonstrates a moderate level of performance, correctly identifying 301 benign samples and 249 malware samples, but with 23 false negatives and 27 false positives, indicating a balanced yet slightly less precise performance compared to other models. The Random Forest model shows superior accuracy with only 9 false positives and 17 false negatives, suggesting that it effectively identifies benign and malware instances with minimal misclassification. The SVM and logistic regression models exhibit comparable performance, both having 14 false positives, with 17 and 13 false negatives, respectively. This indicates reliable classification capabilities,

with logistic regression marginally outperforming SVM in terms of fewer false negatives. Naive Bayes, while still performing reasonably well, has a higher count of false negatives (42) and false positives (20), suggesting that it struggles more with precision and recall compared to other models. XGBoost achieves the best overall performance with only 8 false negatives and 13 false positives, indicating a high level of both precision and recall. This model's confusion matrix reflects its superior capacity to correctly classify both benign and malware samples, showcasing it as the best choice for this classification task.

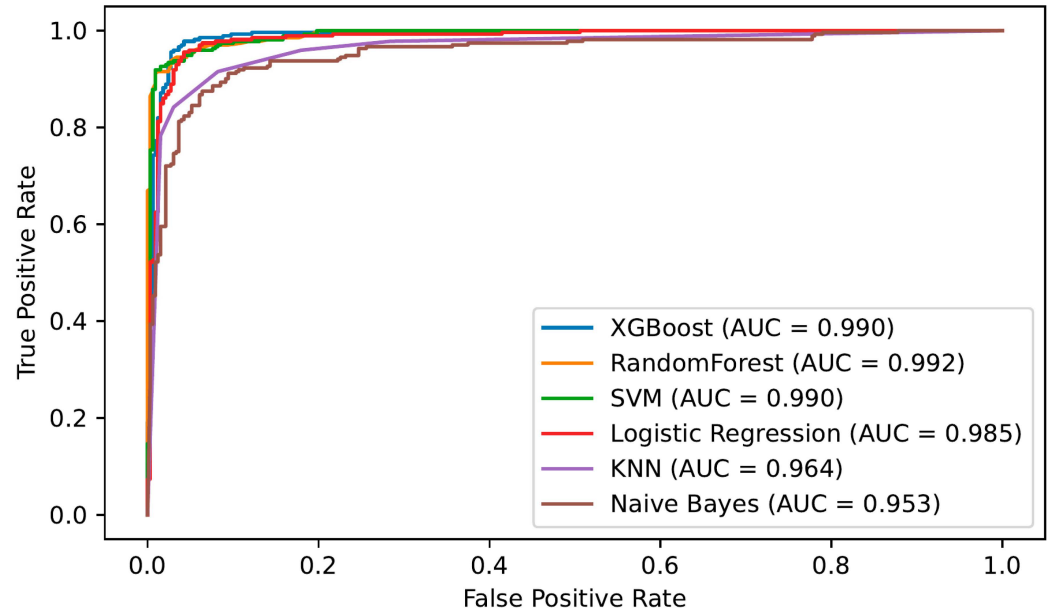


Figure 5. ROC curves comparison of all tested classifiers.

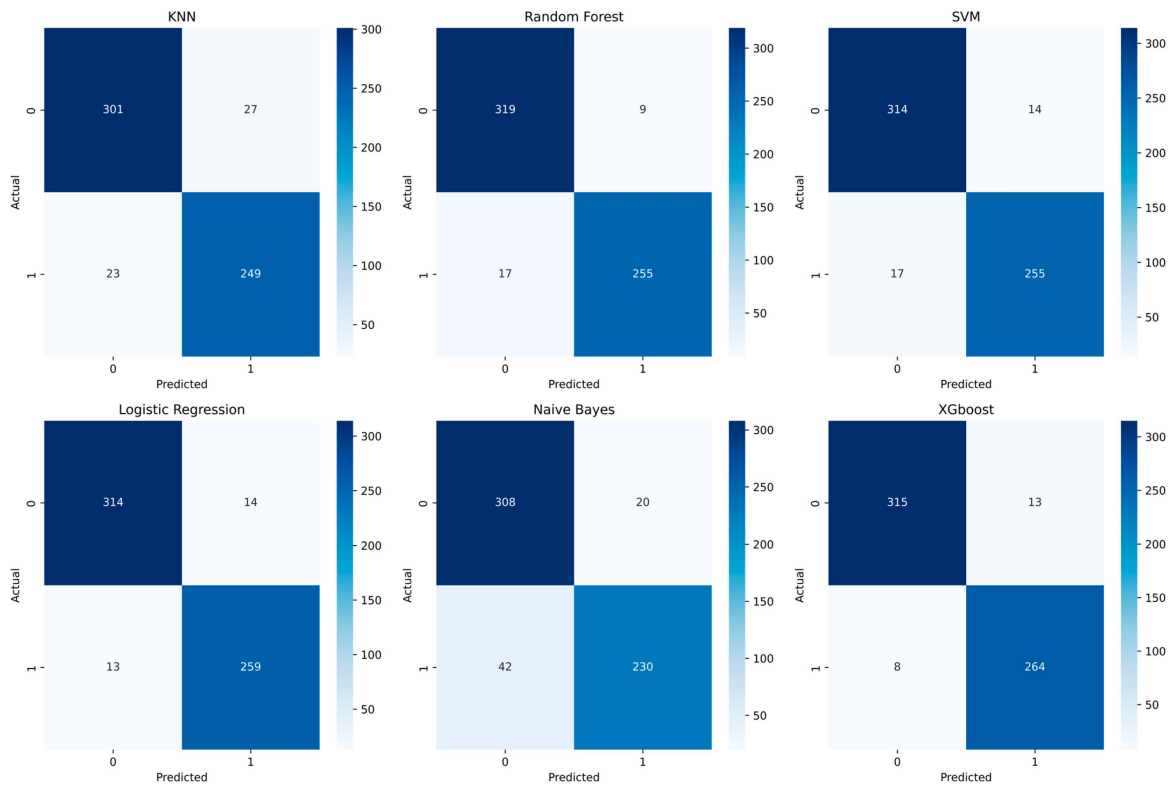


Figure 6. Confusion matrix of all models.

5.2. Feature Importance

In order to understand which features were most influential in the classification of malicious .NET executables, we applied SHAP (SHapley Additive exPlanations), a unified framework for interpreting machine learning models [44]. SHAP assigns each feature an importance score based on how it contributes to individual predictions, making it particularly effective for obtaining global feature importance. We chose SHAP because our goal was to identify the most significant features that impacted the model's predictions at a global level. Unlike simpler feature importance metrics, SHAP considers the interaction between features and provides consistent explanations across different models.

We applied this analysis to the best-performing model, XGBoost, to identify the most impactful features in the classification. Figure 7 presents the global feature importance plot, which displays the top 10 most influential features contributing to the model's decision-making process. The last bar in Figure 7 represents the cumulative SHAP value for the remaining 546 features, indicating their combined effect on the model. The x-axis of Figure 7 represents the mean absolute SHAP values, indicating the magnitude of each feature's contribution to the model predictions. The y-axis lists the top features in descending order of importance. Figure 7 highlights methods such as "system.convert::frombase64string()" and "system.reflection.assembly::load()", which are more frequently used in malicious executables compared to benign ones. The "system.convert::frombase64string()" method is used to obfuscate code or malicious payloads, while "system.reflection.assembly::load()" is used by malware to load additional code (assemblies) dynamically at runtime.

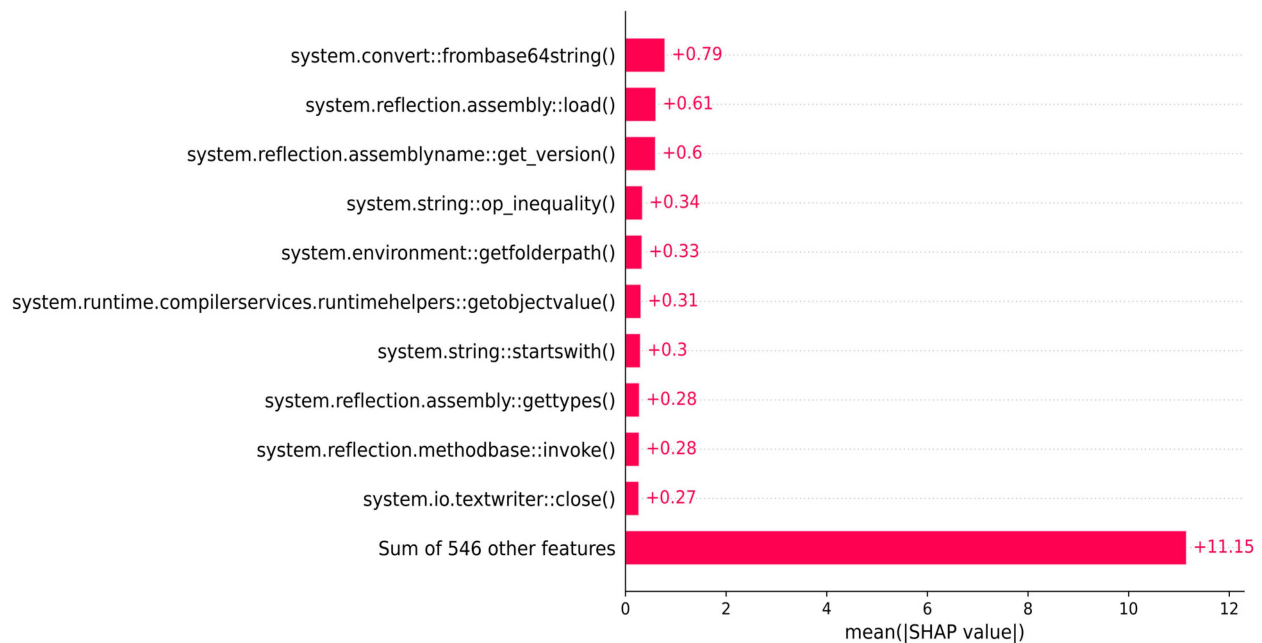


Figure 7. Global feature importance.

Figure 8 shows the SHAP summary plot for the top 10 most important features, which provides a more detailed view of how each feature behaves across the entire dataset. In this plot, each dot represents a single feature value in a row from the dataset. The x-axis represents the SHAP value, indicating whether the feature pushes the prediction toward the positive class (malware) or the negative class (benign). The y-axis lists the features in order of importance. Dots colored in red correspond to high feature values, while blue dots represent low feature values. The insights from Figure 8 are consistent with those from Figure 1. For example, the features "system.convert::frombase64string()" and "system.reflection.assembly::load()" are shown to have higher SHAP values for higher

feature values, meaning that when these methods are used in a sample, they influence the model's prediction towards the malware class (positive class). This alignment across the two figures confirms the importance of these features in driving the classification decision.

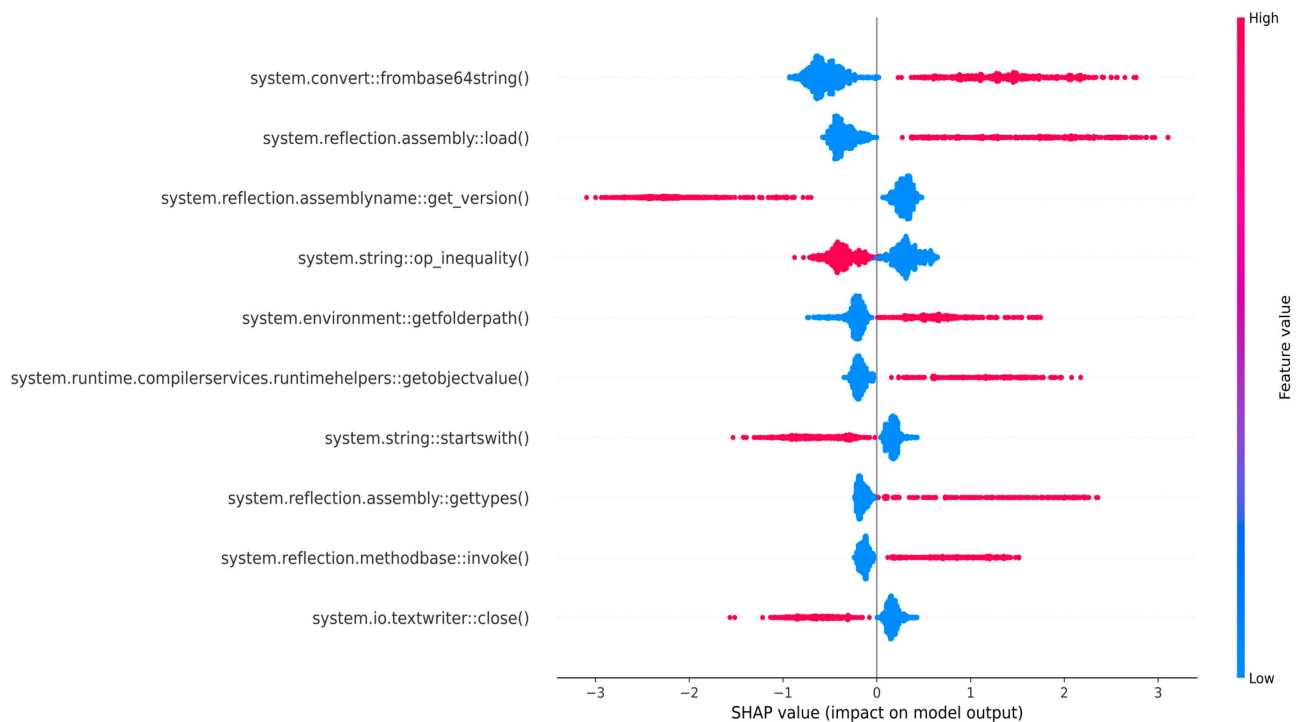


Figure 8. SHAP summary plot for the top 10 features.

5.3. Impact of Feature Length on Classification Performance

In this experiment, we aim to assess the effect of varying feature lengths (method names) on the classification performance. The experiment was structured to systematically adjust feature length and monitor the resulting adjustments in classification metrics, including precision, recall, and F1-score.

Initially, we set a limit of 10 method names per sample as our model. This criterion ensures uniformity across all samples for consistent comparison. If a sample exceeds the ten-method name limit, then an arbitrary selection of features is chosen while setting the remaining ones to zero. This process continues with gradually increasing limits of 40, 60, 80, and finally, 100 features per sample. By following this approach, we systematically explore the impact of varying lengths of feature vectors on our analysis while ensuring a consistent total number of features for each sample. Throughout this experiment, we evaluated the performance of our ML models—XGBoost, random forest, KNN, logistic regression, naïve Bayes, and SVM—on each iteration to compare the impact of varying feature lengths on the classification results. The goal is to systematically analyze how increasing the number of method names (features) influences the accuracy, precision, recall, and F1-score. Table 6 below showcases the evaluation metrics for all four models with varying feature lengths.

The experiment findings showed that increasing feature length generally resulted in higher overall accuracy across all models. Including more method names as features gives machine learning models more information to differentiate and classify malware samples effectively. Both SVM and random forests have a similar pattern in performance improvement as the maximum number of features increases. Specifically, the test set accuracy of both models did not change when increasing the maximum number of features from 80 to 100 for both models. Naïve Bayes's performance is generally lower than that of the other models across all evaluation metrics and feature lengths. However, its test set

accuracy slightly increases as the maximum features length grows. XGBoost consistently outperforms the other models in terms of accuracy when the feature length is greater than 20. Naive Bayes does not show significant improvement with increasing feature length and performs relatively lower than KNN.SVM and random forest demonstrate comparable performance, with random forest showing slightly better results at shorter feature lengths (less than 40) than all modes. This suggests that random forest may be more efficient in handling shorter feature length data as compared to efficient, which appears to rely on longer feature lengths for accurate classification.

Table 6. Evaluation metrics for all five models with different feature lengths.

Model	Features Length	Accuracy	Precision	Recall	F1-Score
XGBoost	20	88.00%	93.85%	78.67%	85.60%
	40	94.33%	95.07%	92.27%	93.65%
	60	94.83%	94.79%	93.75%	94.26%
	80	95.50%	95.53%	94.48%	95.00%
	100	96.00%	95.25%	95.95%	95.60%
Random forest	20	91.00%	96.58%	83.08%	89.32%
	40	94.00%	97.20%	89.33%	93.10%
	60	94.33%	96.12%	91.17%	93.58%
	80	95.00%	96.53%	92.27%	94.36%
	100	95.50%	96.57%	93.38%	94.95%
KNN	20	83.16%	82.75%	79.41%	81.05%
	40	87.50%	87.45%	84.55%	85.98%
	60	89.50%	89.13%	87.50%	88.31%
	80	89.83%	89.51%	87.86%	88.68%
	100	90.50%	89.96%	88.97%	89.46%
SVM	20	88.33%	95.08%	78.30%	85.88%
	40	92.50%	95.21%	87.86%	91.39%
	60	93.33%	94.61%	90.44%	92.48%
	80	94.33%	95.07%	92.27%	93.65%
	100	94.66%	94.77%	93.38%	94.07%
Logistic regression	20	87.04%	93.04%	78.67%	85.25%
	40	93.00%	94.57%	89.70%	92.07%
	60	92.66%	93.84%	89.70%	91.72%
	80	93.50%	94.63%	90.80%	92.68%
	100	94.50%	94.75%	93.01%	93.87%
Naïve Bayes	20	88.83%	90.51%	84.19%	87.23%
	40	89.33%	91.60%	84.19%	87.73%
	60	89.50%	91.63%	84.55%	87.95%
	80	89.66%	92.00%	84.55%	88.12%
	100	89.66%	92.00%	84.55%	88.12%

6. Limitations and Future Work

The main limitation of our methodology is the inability to reliably detect obfuscated or packed .NET malware samples. This is because our approach relies solely on static analysis, which decompiles the malicious sample and extracts method names from the decompiled code. Obfuscation and packing techniques are often used by attackers to avoid detections that rely on static analysis. As a result, our approach may be less effective in detecting certain types of .NET-based malware, particularly those designed to evade detection through obfuscation or packing.

Our framework is designed for binary classification (malware or benign), but in future work, we plan to collect samples from different classes of malware and evaluate machine learning models' ability to accurately classify each malware sample into its respective class.

This will require collecting samples for different classes of malware and fine-tuning machine learning models to achieve high accuracy in multi-class classification tasks. Moreover, we plan to evaluate the effectiveness of deep learning architectures such as recurrent neural networks (RNNs) or long short-term memory (LSTM) on our dataset; this will involve training and testing deep learning architectures on our dataset and assessing their ability to accurately classify malicious and benign .NET executables. Additionally, we also plan to evaluate different tokenization and vectorization methods, such as Word2Vec [45] or word-level tokenization, on the features and assess their impact on the performance of machine learning classifiers. This will allow us to determine which tokenization and vectorization approaches are most effective for our specific dataset and problem domain.

7. Conclusions

This paper proposed a new methodology to detect malicious .NET executables using statically extracted method names. The research created a dataset consisting of these standard method names and evaluated the effectiveness of using these methods to detect .NET malware executable files. The dataset, which is ready for use after the preprocessing, consists of 556 features (.NET method names) and 3000 rows. The dataset consists of 1500 benign and 1500 malware samples. Furthermore, our results indicate that standard .NET methods are reliable features to detect .NET malware through examining them on a variety of machine learning models, such as XGBoost, random forest, KNN, logistic regression, naïve Bayes, and SVM. The results of our experiments demonstrated that XGboost outperformed other models with a test set accuracy of 96.16% and an F1-score of 96.15%. In addition, we confirmed that longer feature lengths enhance the confidence of the classifier's predictions.

Author Contributions: Conceptualization, H.T., R.A. and A.Z.A.; methodology, H.T., R.A., A.A. and A.Z.A.; software, H.T., R.A., A.Z.A. and A.A.A.; validation, R.A., A.A. and A.A.A.; formal analysis, H.T., R.A., A.A. and A.Z.A.; investigation, R.A., A.A. and A.Z.A.; resources, A.Z.A. and A.A.A.; data curation, H.T., R.A. and A.Z.A.; writing—original draft preparation, H.T.; writing—review and editing, R.A., A.A., A.Z.A. and A.A.A.; visualization, H.T.; supervision, R.A. and A.Z.A.; project administration, R.A. and A.Z.A. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The raw data supporting the conclusions of this article will be made available by the authors on request.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. BillWagner. NET Managed Languages Strategy. Microsoft, 6 February 2023. Available online: <https://learn.microsoft.com/en-us/dotnet/fundamentals/languages/> (accessed on 28 March 2024).
2. Maniriho, P.; Mahmood, A.N.; Chowdhury, M.J.M. API-MalDetect: Automated malware detection framework for windows based on API calls and deep learning techniques. *J. Netw. Comput. Appl.* **2023**, *218*, 103704. [CrossRef]
3. D'Angelo, G.; Ficco, M.; Palmieri, F. Malware detection in mobile environments based on Autoencoders and API-images. *J. Parallel Distrib. Comput.* **2020**, *137*, 26–33. [CrossRef]
4. Amer, E.; Zelinka, I. A dynamic Windows malware detection and prediction method based on contextual understanding of API call sequence. *Comput. Secur.* **2020**, *92*, 101760. [CrossRef]
5. Li, C.; Lv, Q.; Li, N.; Wang, Y.; Sun, D.; Qiao, Y. A novel deep framework for dynamic malware detection based on API sequence intrinsic features. *Comput. Secur.* **2022**, *116*, 102686. [CrossRef]
6. Jang-Jaccard, J.; Nepal, S. A survey of emerging threats in cybersecurity. *J. Comput. Syst. Sci.* **2014**, *80*, 973–993. [CrossRef]

7. Mani, G.; Sethuraman, S.C. A comprehensive survey on deep learning based malware detection techniques. *Comput. Sci. Rev.* **2023**, *47*, 100529. [[CrossRef](#)]
8. Shankarapani, M.K.; Ramamoorthy, S.; Movva, R.S.; Mukkamala, S. Malware detection using assembly and API call sequences. *J. Comput. Virol.* **2011**, *7*, 107–119. [[CrossRef](#)]
9. Introducing .NET Assemblies. In *Pro VB 2008 and the .NET 3.5 Platform*; Apress: Berkeley, CA, USA, 2008; pp. 437–481. [[CrossRef](#)]
10. Troelsen, A.; Japikse, P. Understanding CIL and the Role of Dynamic Assemblies. In *Pro C# 8 with .NET Core 3*; Apress: Berkeley, CA, USA, 2020; pp. 661–696. [[CrossRef](#)]
11. Rabadi, D.; Teo, S.G. Advanced Windows Methods on Malware Detection and Classification. *Assoc. Comput. Mach.* **2020**, 54–68. [[CrossRef](#)]
12. Pistelli, D. The .NET File Format. CodeProject. Available online: <https://www.codeproject.com/Articles/12585/The-.NET-File-Format> (accessed on 4 August 2024).
13. Richter, J. *Applied Microsoft: .NET Framework Programming*; Microsoft Press Redmond: Redmond, WA, USA, 2002; Volume 1.
14. Zhang, S.; Wu, J.; Zhang, M.; Yang, W. Dynamic Malware Analysis Based on API Sequence Semantic Fusion. *Appl. Sci.* **2023**, *13*, 6526. [[CrossRef](#)]
15. Shin, K.; Lee, Y.; Lim, J.; Kang, H.; Lee, S. System API Vectorization for Malware Detection. *IEEE Access* **2023**, *11*, 53788–53805. [[CrossRef](#)]
16. Cui, L.; Cui, J.; Ji, Y.; Hao, Z.; Li, L.; Ding, Z. API2Vec: Learning Representations of API Sequences for Malware Detection. In Proceedings of the ISSTA 2023—Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, Seattle, WA, USA, 17–21 July 2023; pp. 261–273. [[CrossRef](#)]
17. Prachi; Dabas, N.; Sharma, P. MalAnalyser: An effective and efficient Windows malware detection method based on API call sequences. *Expert Syst. Appl.* **2023**, *230*, 120756. [[CrossRef](#)]
18. Almousa, M.; Basavaraju, S.; Anwar, M. API-Based Ransomware Detection Using Machine Learning-Based Threat Detection Models. In Proceedings of the 2021 18th International Conference on Privacy, Security and Trust, PST 2021, Auckland, New Zealand, 13–15 December 2021. [[CrossRef](#)]
19. Mathew, J.; Kumara, M.A.A. API call based malware detection approach using recurrent neural network—LSTM. In *Advances in Intelligent Systems and Computing*; Springer: Berlin/Heidelberg, Germany, 2020; pp. 87–99. [[CrossRef](#)]
20. Catak, F.O.; Yazi, A.F.; Elezaj, O.; Ahmed, J. Deep learning based Sequential model for malware analysis using Windows exe API Calls. *PeerJ Comput Sci* **2020**, *6*, e285. [[CrossRef](#)] [[PubMed](#)]
21. “0xd4d/dnlib.” 0xd4d, 29 February 2024. Available online: <https://github.com/0xd4d/dnlib> (accessed on 3 March 2024).
22. Abujayyab, S.K.M.; Almajalid, R.; Wazirali, R.; Ahmad, R.; Taşoğlu, E.; Karas, I.R.; Hijazi, I. Integrating object-based and pixel-based segmentation for building footprint extraction from satellite images. *J. King Saud Univ.—Comput. Inf. Sci.* **2023**, *35*, 101802. [[CrossRef](#)]
23. Ahmad, R. Smart remote sensing network for disaster management: An overview. *Telecommun. Syst.* **2024**, *87*, 213–237. [[CrossRef](#)]
24. Liu, G.; Zhao, H.; Fan, F.; Liu, G.; Xu, Q.; Nazir, S. An Enhanced Intrusion Detection Model Based on Improved kNN in WSNs. *Sensors* **2022**, *22*, 1407. [[CrossRef](#)]
25. Zidi, S.; Moulahi, T.; Alaya, B. Fault detection in wireless sensor networks through SVM classifier. *IEEE Sens J* **2018**, *18*, 340–347. [[CrossRef](#)]
26. Dener, M.; Ok, G.; Orman, A. Malware Detection Using Memory Analysis Data in Big Data Environment. *Appl. Sci.* **2022**, *12*, 8604. [[CrossRef](#)]
27. Maniriho, P.; Mahmood, A.N.; Chowdhury, M.J.M. EarlyMalDetect: A Novel Approach for Early Windows Malware Detection Based on Sequences of API Calls. *arXiv* **2024**, arXiv:2407.13355. [[CrossRef](#)]
28. Ahmed, M.; Afreen, N.; Ahmed, M.; Sameer, M.; Ahamed, J. An inception V3 approach for malware classification using machine learning and transfer learning. *Int. J. Intell. Netw.* **2023**, *4*, 11–18. [[CrossRef](#)]
29. Manna, M.; Case, A.; Ali-Gombe, A.; Richard, G.G. Memory analysis of .NET and .Net Core applications. *Forensic Sci. Int. Digit. Investig.* **2022**, *42*, 301404. [[CrossRef](#)]
30. Or-Meir, O.; Nissim, N.; Elovici, Y.; Rokach, L. Dynamic malware analysis in the modern era—A state of the art survey. *ACM Comput. Surv.* **2019**, *52*, 1–48. [[CrossRef](#)]
31. Souri, A.; Hosseini, R. A state-of-the-art survey of malware detection approaches using data mining techniques. *Hum.-Centric Comput. Inf. Sci.* **2018**, *8*, 3. [[CrossRef](#)]
32. Dick, J.R.; Kent, K.B.; Libby, J.C. A partitioning analysis of the .NET common language runtime. In Proceedings of the International Symposium and Workshop on Engineering of Computer Based Systems, Tucson, AR, USA, 27–29 November 2007; pp. 317–323. [[CrossRef](#)]
33. Microsoft. What Is .NET Framework? A Software Development Framework. Available online: <https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet-framework/> (accessed on 28 March 2024).
34. ARM. Arm Architecture. Available online: <https://www.arm.com/architecture> (accessed on 4 August 2024).
35. MalwareBazaar. MalwareBazaar | Malware Sample Exchange. Available online: <https://bazaar.abuse.ch/> (accessed on 3 March 2024).

36. VirusShare. VirusShare.com. Available online: <https://virusshare.com/> (accessed on 3 March 2024).
37. SourceForge. Compare, Download & Develop Open Source & Business Software—SourceForge. Available online: <https://sourceforge.net/> (accessed on 3 March 2024).
38. GitHub. GitHub: Let's Build from Here. GitHub. Available online: <https://github.com/> (accessed on 3 March 2024).
39. Ventura, E.C. Pefile. 2023. Available online: <https://github.com/erocarrera/pefile> (accessed on 3 March 2024).
40. Galal, H.S.; Mahdy, Y.B.; Atia, M.A. Behavior-based features model for malware detection. *J. Comput. Virol. Hack Tech.* **2016**, *12*, 59–67. [[CrossRef](#)]
41. Banin, S.; Dyrkolbotn, G.O. Multinomial malware classification via low-level features. *Digit. Investig.* **2018**, *26*, S107–S117. [[CrossRef](#)]
42. Syeda, D.Z.; Asghar, M.N. Dynamic Malware Classification and API Categorisation of Windows Portable Executable Files Using Machine Learning. *Appl. Sci.* **2024**, *14*, 1015. [[CrossRef](#)]
43. Singh, J.; McCann, B.; Socher, R.; Xiong, C. BERT is Not an Interlingua and the Bias of Tokenization. In Proceedings of the 2nd Workshop on Deep Learning Approaches for Low-Resource NLP (DeepLo 2019), Hong Kong, China, 3 November 2019; pp. 47–55. [[CrossRef](#)]
44. Lundberg, S.; Lee, S.-I. A Unified Approach to Interpreting Model Predictions. *arXiv* **2017**, arXiv:1705.07874. [[CrossRef](#)]
45. Mikolov, T.; Chen, K.; Corrado, G.; Dean, J. Efficient Estimation of Word Representations in Vector Space. *arXiv* **2013**, arXiv:1301.3781. Available online: <http://arxiv.org/abs/1301.3781> (accessed on 3 March 2024).

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.