A Novel Feature Vector for AI-assisted Windows Malware Detection

Le Qi Yau*, Yik Ting Lam†, Ashwin Lokesh‡, Prannaya Gupta§, Justin Lim§, Ishneet Sukhvinder Singh*, Jia-Yi Loo¶, Mao V. Ngo‖, Sin G. Teo¶, Tram Truong-Huu**
*Temasek Junior College, Singapore, †Dunman High School, Singapore, ‡Yishun Innova Junior College,
§NUS High School of Math and Science, Singapore, ¶Agency for Science, Technology and Research, Singapore
‖Singapore University of Technology and Design, Singapore, **Singapore Institute of Technology, Singapore
Email: yau_le_qi@temasekjc.moe.edu.sg, oneytlam@gmail.com, ashlokpry@gmail.com, prannayagupta@gmail.com,
limsuehan@gmail.com, ishneet_singh@temasekjc.moe.edu.sg, loojy@i2r.a-star.edu.sg,
vanmao_ngo@sutd.edu.sg, teosg@i2r.a-star.edu.sg, truonghuu.tram@singaporetech.edu.sg

***Abstract*—Dynamic malware analysis, which has been a major field in malware analysis and detection, involves executing malware in a controlled environment and observing its behavior. Dynamic analysis reports include Windows API calls, which are extracted as a data source for statistical features, and have allowed for effective malware detection. However, existing works neglect certain critical information about the API calls when constructing feature vectors. In this work, we develop a novel feature vector, taking into account not only the API name and its arguments but also other statistical features such as the return values and the number of times it is called in a sample. Due to the diversity of API calls in terms of the number of arguments, names, and return values, we adopt hash functions to construct a fixed-size feature vector, thus facilitating the design and development of artificial intelligence (AI)-assisted algorithms for malware detection. We experiment with various deep learning and machine learning models and perform extensive hyperparameter tuning to come up with an optimal model for our feature vector. The experimental dataset was recently collected from an anti-virus company, including 14860 samples with 7398 malign samples and 7462 benign samples. Extensive experiments show that our solution outperforms many baseline state-of-the-art malware detectors in various performance metrics, including accuracy, and false positive or false negative rate, thus proving the effectiveness of our feature vector and detection models.**

***Index Terms*—Malware Analysis and Detection, Feature Engineering, Machine Learning, Deep Learning.**

I. Introduction

Malware is defined as malicious software such as viruses, worms, Trojans, adware, Backdoor, etc., specifically developed to perform malicious activities in target systems without the user's knowledge such as deleting/encrypting files or stealing sensitive information. In recent times, malware has evolved greatly with ransomware such as WANNACRY costing an estimated 4 billion dollars in damage [1]. It is therefore of utmost importance that there is a reliable mechanism that could detect the presence of malware to prevent them from entering the computer and wreaking havoc.

Over the years, researchers have developed two approaches to analyzing malware. Static malware analysis involves extracting static properties of the malware file such as headers, metadata, embedded assets, etc. [2]. These properties can then be hashed into a signature and compared to existing signatures to deduce whether the file is malicious or not. However, static malware analysis is weak to code obfuscation [3] and is also unable to detect "zero-day" malware [4], [5]. In contrast, dynamic analysis is carried out by executing the malware in a sandbox environment, such as the Cuckoo sandbox, to analyze its behavior. Dynamic analysis has been shown to be less vulnerable to techniques such as code obfuscation [6]. As such, we continue advocating dynamic analysis in this paper to enhance the performance of malware detection systems.

Based on the dynamic analysis reports, various techniques have been developed to extract dynamic features, which are used to train a machine learning or deep learning model for malware detection [5], [7]–[10]. The main difference among these works is the way of construction of the feature vectors, which in turn affects the design and performance of machine learning or deep learning models used for detection. For instance, the work presented in [5] extracts the sequence of Windows API calls combined with their arguments of malware samples, which is fed into a long short-term memory (LSTM) neural network. However, the sequence can be easily obfuscated as adversaries can change the order of Windows API calls in the sequence without changing the semantic (malicious behavior) of samples. Other work such as [8] also extracts API calls and their arguments before constructing a feature vector that indicates the occurrence of a feature (i.e., a combination of an API call and its arguments) in a sample. This approach makes the size of the feature vector depend on the dataset and its diversity, thus limiting the generalization of the approach in practice.

In this work, we address the above challenges by developing a novel feature vector of a fixed size suitable for various machine learning or deep learning models. Apart from API calls and their arguments, we propose to take into account further information that can be extracted from the dynamic analysis reports but has not been used in previous works. For instance, the return values of the API calls are useful information that can also indicate whether the activity is suspicious or not. An unsuccessful file creation activity (with

API `NtCreateFile`) may indicate that it is suspicious as the running user does not have permission to create files in a specific location, e.g., "`C:\Windows\system32`". We propose to combine the extracted information of API calls and their statistics into a comprehensive feature vector. The main contribution of the paper is as follows:

- A novel feature vector that takes into the API name, arguments, return values, and the total number of occurrences of the API, combined with the adoption of hash functions to enable the independence of sample heterogeneity on the length of the feature vector.
- An extensive experiment with various machine learning algorithms and deep learning models (in short AI models) for an empirical comparison among the models and to draw a conclusion on the model with the best performance for our novel feature vector.
- A performance comparison with existing works to demonstrate the effectiveness of the developed feature vector and AI algorithms.

The remainder of the paper is organized as follows. Section II reviews the literature on malware analysis and detection. Section III presents our novel feature vector and AI models used for malware detection. Section IV describes the dataset used for experiments and performance analysis. We conclude our work in Section V.

II. Related Works

In this section, we review the literature on malware analysis and detection using machine learning and deep learning. The existing works differentiate from each other by (i) the approach to extraction and construction of feature vectors, and (ii) the machine learning or deep learning models used for detection. In [11] and [12], the authors proposed to convert each malware sample into an image (every byte of the sample is converted into a number between 0 and 255) and then apply convolutional neural networks for detection. In [13], Ndibanje *et al.* developed a hybrid technique to detect Windows malware samples. The technique performs similarity analysis to classify malware based on distance measures including Minkowski Distance, Cosine Similarity, Containment Broder, Canberra distance, and Longest CommonSubsequence (LCS). In [10], the authors proposed to extract API calls not only through dynamic analysis but also through static analysis. Based on static and dynamic API sequences, a mapping is performed to produce a hybrid API sequence for each sample, which is used to train machine learning models for malware detection. The above works rely only on the sequence of the API calls and ignore the API arguments. In [14], the authors generated two-pixel vectors using color-coded features out of the static-based byte frequency information and dynamic-based API call information. The two-pixel vectors are then reshaped into two-pixel matrices and saved as two images to be incorporated later as one image and used to train a VGG16 network model.

Zhang *et al.* [5] introduced a feature vector by using hash functions to manipulate various information, including the name, category, and arguments of the API calls. The feature vector is then fed into a multi-gated convolutional neural network to reduce the high-dimensional hash features from each API call. The extracted features are finally fed into a Bidirectional Long Short-Term Memory (LSTM) neural network to learn the sequential correlation between API calls. Chen *et al.* [15] proposed a method where rule-based and cluster-based classifications are used to determine how vulnerable a parameter of an API is to malicious behavior. The API is then labeled based on its parameters with different levels of sensitivity. The API is encoded by combining the sensitivity and native embedding of the labeled API, to determine the sequential correlation between API calls and the relationship of the security semantics between the calls. The encoded API is finally passed through a deep neural network to classify the sample whether it is malicious or not. In [16], an in-house hooking tool has been developed to study the sequence of API calls (only a subset of 126 APIs), their arguments, with and without their return values (API-ARG or API-ARG-RET) extracted during the execution of PE files in Windows XP OS. The feature sets are used to train different machine learning algorithms to detect malicious from benign files. In [17], Zhang *et al.* proposed a deceptive engine called Scarecrow to exploit evasion techniques of malware samples. The authors proposed to transform or camouflage the physical end-user environment into an analysis-like environment from the view of evasive malware.

Rabadi *et al.* [8] developed two methods to construct the feature vectors based on the Windows API calls and their arguments extracted from the dynamic analysis reports. The first method combines an API call with every single argument to create a feature value. The number of features of each sample is the sum of the number of arguments of each API call invoked by the sample. The second method combines each API call with all its arguments to create a feature value. The number of features of each sample is therefore the number of API calls invoked by the sample. Due to the heterogeneity of malware, the resulting feature vector is sparse with a size of 2^{20}. Tran *et al.* [18] used traditional Natural Language Processing methods on the API sequences, such as TF-IDF, Paragraph Vector with Distributed Bag of Words, or Distributed Memory. The extracted feature is then passed through a machine learning model for detection. The authors have experimented with various models such as Support Vector Machine, k-nearest neighbors, Multi-layer Perception, and Random Forest. Damodaran *et al.* [19] trained Hidden Markov Models (HMMs) on combined static and dynamic features. Their results present the dynamic set as the most optimal feature set for the HMM. Existing approaches do not take into account the return value of API calls, which could indicate the success of its execution. We address the above challenges by developing a novel feature vector in this work.

III. Methodology

A. Overview

The overview of the AI-assisted malware analysis and detection framework is presented in Fig. 1. The framework

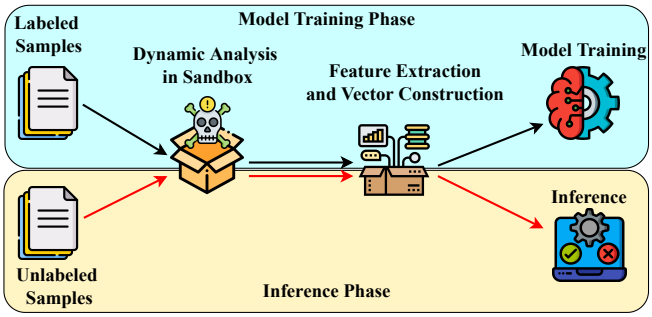

Fig. 1: AI-assisted malware analysis and detection framework.

is composed of two phases: a model training phase and an inference phase. In the beginning, when there is no model available, the model training phase has to take place first to provide a trained model for the inference phase. At a later time when a trained model has been deployed for inference, these two phases can run in parallel such that the model training phase keeps updating or training a new model with new/unseen samples. The new model will be deployed to replace the model in the inference phase if there is a performance drop.

To perform dynamic analysis of samples, we employ the Cuckoo sandbox[1], an open-source sandbox. It is to be noted that dynamic analysis of samples in sandboxes is a time-consuming task. We use our parallel and distributed dynamic analysis platform developed in our previous work [6] to speed up the analysis. In this platform, a server coordinates multiple virtual machines (VM) hosting Cuckoo sandboxes (up to 20 VMs) to perform dynamic analysis. Given a bundle of samples, the server concurrently sends samples to the VMs and monitors their execution. the platform supports fault tolerance such that if a VM crashes, the server will decommission the VM and instantiate a new one before resubmitting the sample for a repeat analysis. We implement a feature extraction and construction module that incorporates various elements of the API calls such as the API name, category, arguments, return value, and API call frequency. Due to the heterogeneity of names and arguments of Windows API calls, we adopt the hashing approach [20] to encode extracted features into a fixed-size vector. Given a set of dynamic analysis reports of both malicious samples and benign samples, we use the feature extraction and vector construction module to process the reports and produce a labeled dataset, which in turn is used for training detection models.

The feature vector produced by the feature extraction and construction module is generic and can be used by many machine learning or deep learning models for inference. In this work, we employ several machine learning models such as LightGBM [21], Random Forest (RF) [22], XGBoost [23], etc. We also adopt deep learning models especially 1-dimensional convolution neural networks, which can learn important features and generate the latent vector for each sample. The latent vector is then fed into a Multi-layer Perceptron (MLP) to determine whether a sample is malicious or benign.

[1]Cuckoo sandbox: https://cuckoo.cert.ee

TABLE I: Feature Details

Feature Types			**Details**	**Dim.**
API Name	Strings		Internal Words Hashing Trick	8
API Category	Strings		Hashing Trick	4
API Arguments	Integers		Hashing Trick	16
	Strings	Paths	Hashing Trick with Hierarchy	16
		DLLs		8
		Registry Keys		12
		URLs		16
		IPs		12
		String Statistics	numbStrings, avLength, numbChars, entropy, numbPaths, numbDlls, numbUrls, numbIPs, numbRegistryKeys, numbMZ	10
Return Values			Hashing Trick	2
Global Statistics			numbCalls	1
Total Dimension				**105**

B. Feature Extraction

Cuckoo sandbox is able to hook more than 300 Windows APIs to perform dynamic analysis [24]. Each sample is executed in a Windows 7 Guest OS deployed in the Cuckoo sandbox, which uses hooking techniques to intercept the execution and collect the runtime behavior of the samples. The output of the dynamic analysis is a JSON file that includes various information about the execution of the sample in the guest OS. The feature extraction module goes through the JSON file to extract all API calls with their arguments and return values. Since many APIs have the same name with a different suffice due to extension and backward compatibility of Windows OS (e.g., `Ex`, `A`, `W`, `ExA`, `ExW`), we remove the suffices and perform merging. For instance, `FindFirstFileExW` and `FindFirstFileExA` have the same functionality and are merged into `FindFirstFile`. After merging suffices, the number of supported APIs is reduced to 258.

C. Feature Vector Construction

We now present the feature vector construction technique based on the hashing trick approach [20], which has also been used in [5]. For each API, we construct a feature vector with a size of 105 elements as shown in Table I. The last column indicates the number of bins in the hashing trick, which is empirically determined through extensive experiments. The higher the number of bins, the less the collision happening but it may also create a sparse feature vector. We also note that an API may be called multiple times in a sample, each time being with different arguments, and it may return a different value. Thus, processing the API arguments and return values is more challenging.

1) API Name and Category: An API name is a string composed of multiple words, each having the first letter capitalized, e.g., `FindFirstFile`. We divide the API name into an array of words and apply the hashing trick on each of the words. Cuckoo classifies the APIs into 17 categories whose name is a single word. For instance, Listing 1 shows a snippet of the collected API call `LdrGetDllHandle` classified in the `system` category during the execution of the sample with the `SHA1` value of `6d1b9a54aa31d79b86f5acff07bc4b323b59e108`. We divide the words of the categories into characters and apply the hashing function. Given a sequence of characters or words for an API name or category (denoted as x), we apply the following equation to transform the sequence into a fixed-size vector.

$$\phi_i(x) = \sum_{j:h(x_j)=i} \xi(x_j) \tag{1}$$

where $\phi_i(x)$ is the value of the i-th bin, $i = 1...K$ and K is the number of bins, e.g., 8 for an API name and 16 for a path in API arguments. $h(x_j)$ is the hash function that maps a value (x_j) to the bin index. It is to be noted that we use two hash functions for the API name and category respectively since the number of bins for the API names is different from that of the API categories. $\xi(x_j)$ is another hash function that maps a value x_j to $\{\pm 1\}$. In other words, we add $\xi(x_j)$ to the bin.


```
{
    "category": "system",
    "status": 0,
    "stacktrace": [],
    "last_error": 0,
    "nt_status": -1073741515,
    "api": "LdrGetDllHandle",
    "return_value": 3221225781,
    "arguments": {
        "module_name": "VERSION.DLL",
        "stack_pivoted": 0,
        "module_address": "0x00000000"
    },
    "time": 1677671645.633939,
    "tid": 8844,
    "flags": {}
}
```


Listing 1: API call with their category recorded by Cuckoo Sandbox in JSON format.

2) API Arguments and Return Values: Unlike previous works (such as [5]) which do not consider the return values of API calls, we believe that this information is important as it indicates whether the activity is successful or not. While the fact that an API call successfully executes does not say much, a failed execution, especially those linked with access permission could indicate that the users are not allowed to perform those actions and the action is suspicious.

API arguments and return values can be categorized into two classes: numerical values and strings. Cuckoo dynamic analysis reports additionally provide the name of each API argument. For API arguments and return values that are numerical, rather than using the value itself as a feature, we apply the following equation to incorporate the argument name into the feature vector to provide more meaning. We also apply the logarithm to reduce the sparsity of the argument or return values in the resulting vector.

$$\phi_i(x) = \sum_{j:h(x_j^{\text{name}})=i} \xi(x_j^{\text{name}}) \log(|x^{\text{value}}| + 1) \tag{2}$$

where h and ξ are the two hash functions presented in Eq. (1) with a different number of bins (i.e., 16). x is defined as a tuple $(x^{\text{name}}, x^{\text{value}})$ and x_j^{name} is an element of the name sequence (x^{name}) after splitting. For the APIs that are called multiple times in a sample, we repeatedly process the arguments and return values of each call and update the bins via summation.

For argument or return values with the string type, we deliberately analyze some important types: paths, DLLs, registry keys, URLs, and IPs, which could contain indicators of compromises [25]. We keep using Eq. (1) to convert these strings into fixed-size feature vectors. For each type of string, we define the number of bins of the hash function accordingly as shown in Table I. It is also to be noted that to capture the hierarchical information, these strings are parsed into multiple sub-strings before applying the hashing trick. The difference is the splitting delimiter. For instance, a string argument-value "`D:\dir\subdir\subsubdir`" is recognized as a file path, and it is decomposed into four sub-strings including "`C:\`", "`D:\dir`", "`D:\dir\subdir`", and "`D:\dir\subdir\subsubdir`". Other strings such as IP addresses and URLs are decomposed with dots as the delimiter. For URLs, we only use the hostname as most web servers use HTTPS as the communication protocol.

Apart from API arguments and return values, we additionally consider their statistical information and add them to the feature vector of each sample. We capture 10 statistical values for Windows API arguments and return values including the number of strings, the average length of the strings, the number of characters in each string, the entropy of characters across all strings, the number of DLLs, register keys, IPs, "MZ" strings, and paths. Last but not least, we also consider the number of times the API is called in a sample, which has solely been used as a feature vector (API frequency) in previous works [26]. With a total of 258 Windows APIs, we end up with a feature vector that has $258 \times 105 = 27090$ elements. We note that a malware sample may not use all 258 Windows APIs in its codes. For the APIs which are not used, a vector of zeros is included. It is also to be noted that we do not consider the order of APIs used in the samples as we discussed in the introduction about its weakness against code obfuscation. Instead, we fix an order that is consistent for any samples.

D. Machine Learning and Deep Learning Models

Given the feature vector described in the previous section, various machine learning and deep learning models can be used. Tree-based algorithms such as LightGBM [21], RF [22] and XGBoost [23] have demonstrated their capability in binary classification. We adopt these algorithms to our problem. RF constructs a multitude of decision trees at training time and applies the ensemble learning method to determine the class of

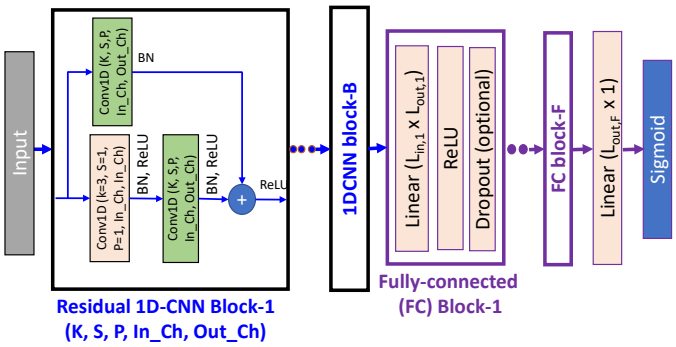

Fig. 2: Architecture of ResNet1D with Multiple 1D-CNN Residual Blocks.

TABLE II: Detailed architecture of ResNet1D model

Block	Kernel size	Stride	Padding	In : Out channels
conv1	12	3	0	1 : 32
conv2	11	4	0	32 : 64
conv3	11	4	0	64 : 128
conv4	12	5	1	128 : 256
conv5	11	5	0	256 : 512
conv6	11	5	0	512 : 1024
conv7	3	1	0	1024 : 2048
FC1	2048 x 1024			
FC2	1024 x 1			
Sigmoid	1			

the input sample. XGBoost and LightGBM are scalable tree-boosting systems that are suitable for sparse feature vectors.

With the emergence of deep learning, residual deep neural networks (ResNet) [27] with convolutional neural network (CNN) blocks have demonstrated their capability in other disciplines such as computer vision. On one hand, we adopt existing models such as EfficientNet [28]. On the other hand, we develop ResNet1D from the original ResNet model by replacing the residual Conv2D blocks with residual Conv1D blocks. The architecture of ResNet1D is depicted in Fig. 2. The detailed parameters of the model are given in Table II including kernel size, stride, padding, and output channels, which are denoted as K, S, P, Out_Ch as shown in Fig. 2. The number of input channels (In_Ch) of the first residual Conv1D block is the number of elements in the input feature vector (27090). From the second block onwards, the number of input channels is the same as the number of output channels of the precedent block. We advocate one-dimensional convolution neural networks (1D-CNN) whose convolutional filters are used to identify patterns or features within the tensors that are relevant to the task at hand. The rationale for using a 1D-CNN is that our feature vector with 27090 elements does not have two-dimensional correlations like images. Obviously, we can shape the input feature vector as a 2D matrix of the size of 258×105, then apply a 2D-CNN model for capturing intra-feature correlation (i.e., within a line of 105 features for a particular API) and inter-feature correlation (i.e., cross neighboring features). However, to avoid learning the order of APIs to evade obfuscation, we flatten the input and use a 1D-CNN instead of a 2D-CNN. We train ResNet1D for

TABLE III: Detailed parameters of MLP model

Layer	Type	Activation	Dropout	Input x Output size
Input	Linear	ReLU	0.3	27,090 x 10,000
Hidden	Linear	ReLU	0.3	10,000 x 5,000
Hidden	Linear	ReLU	0.3	5,000 x 2,500
Hidden	Linear	ReLU	0.4	2,500 x 1,000
Hidden	Linear	ReLU	0.4	1,000 x 500
Hidden	Linear	ReLU	0.5	500 x 250
Hidden	Linear	ReLU	0.5	250 x 1
Output	Non-linear	Sigmoid	-	1 x 1

100 training epochs, using stochastic gradient descent (SGD) optimizer with a momentum of 0.9. The initial learning rate is set as 0.02 and reduced by 10 times at the 50^{th} epoch. We also use a weight-decay of 0.001 as a regularizer during the model training process. Before training ResNet1D, we do preprocess data by standardizing the training data to zero mean and unit variance along 105 features, then reshape the input data to [27,090 x 1 channel] before feeding into the Conv1D blocks.

We additionally design a multi-layer perceptron (MLP), whose architecture is detailed in Table III. The model includes several fully connected (linear) layers and ends with an output layer with a `Sigmoid` function to indicate whether a sample is malicious or not. We set the batch size parameter to 128 and train the model for 100 epochs. We use Adam optimizer with an initial learning rate of `1e-4` and multistep scheduler to reduce the learning rate at the 45^{th} and 60^{th} epochs (learning rate is reduced by $\gamma = 0.1$), and weight decay of `1e-5`. Similar to the preprocessing step we used for training ResNet1D, we standardize data along 105 features, then reshape the input to [27,090] dimensions.

IV. Experiments

A. Experimental Settings

1) Datasets: Our dataset was obtained from SecureAge[2], a local cybersecurity company based in Singapore. The dataset contains 14860 samples including 7462 samples that are benign and 7398 samples that are malign. All the samples were collected from July 2019 to December 2019. Malicious samples include more than 20 malware types from popular types such as Trojan, Virus, Worm, Adware, and Backdoor to less popular types such as Exploit, Rootkit, NetTool, and Eicar. We applied 5-fold cross-validation when evaluating model performance and took the mean metrics to ensure that the model is not biased due to the randomness in the train-test splitting of the dataset.

2) Performance Metrics: We evaluate the performance of the models using Accuracy, Precision, Recall, Area Under Curve (AUC-ROC), False Positive Rate (FPR), and False Negative Rate (FNR). The Recall of the model represents the ratio of the accurately detected malicious samples over all malicious samples while the FPR represents the ratio of benign samples classified inaccurately as malicious. A high Recall indicates that the model accurately classifies most malicious samples while a low FPR means that the model does not

[2]SecureAge: https://www.secureage.com/

frequently wrongly classify benign samples as malicious. In contrast, FNR indicates the number of malicious samples classified as benign. In this case, the misclassified malicious samples can compromise the host and cause damage. As such, we aim to achieve a high Recall and a low FNR as well as FPR for the proposed models. We note that FNR and FPR are two orthogonal metrics. Achieving a low value for both metrics is challenging. Thus, COTS products often sacrifice FPR to achieve a low FNR.

3) Existing Methods for Performance Comparison: To evaluate the performance of the proposed feature vector, we compare the performance of the model trained with the proposed feature vector to the following existing approaches:

- Zhang *et al.* [5]: This work extracts API calls and their arguments to construct the feature vectors and develop a deep learning model that combines multiple gated CNNs and a bidirectional LSTM for malware detection. This work does not consider the return values and global statistics of each API.
- Rabadi *et al.* [8]: Similar to [5], this work also extracts API calls and their arguments but developed two approaches to construct the binary feature vectors. The first approach is to combine the API name and all its arguments to form a feature value. For instance, the API presented in Listing 1 produces a feature value that is the hash value of the string "`LdrGetDLLHandle:0=VERSION.DLL;LdrGetDLLHandle:1=0;LdrGetDLLHandle:2=0x00000000`". The second approach is to combine the API with each of the arguments to form a feature value. For instance, the example shown in Listing 1 creates 3 feature values represented by three strings: "`LdrGetDLLHandle:0=VERSION.DLL`", "`LdrGetDLLHandle:1=0`", and "`LdrGetDLLHandle:2=0x00000000`". The entire training data is processed, creating a large set of features. Each sample will be represented by a binary vector indicating whether the sample has a feature value or not. The feature vector is passed through conventional machine learning algorithms such as XGBoost and Random Forest. We compare our model against XGBoost as it has the highest performance reported in this work.

For both state-of-the-art methods, we request the source codes of the methods from the authors and rerun the experiments with our dataset including feature extraction, model training, and testing. This allows us to perform a fair performance comparison among the methods.

B. Analysis of Results

1) Comparison with existing methods: As seen in Table IV, our model outperforms existing methods in all performance metrics. Precisely, our model achieves an improvement of up to 0.9% in accuracy and up to 2.03% in Recall. This is explained by the fact that Zhang *et al.*'s approach constructs feature vectors based on the sequence of API calls and their arguments, which is vulnerable to code obfuscation. Rabadi *et al.*'s feature vector is very sparse with many zeros due to the diversity of API arguments. The vector size is also very large (up to 2^20 in our experiments) and depends on the training dataset. If we narrow down to the FNR, which is an important metric of malware detection systems, we can observe that for every 100 malicious samples, Zhang *et al.*'s technique misclassifies almost 5 malicious samples while our technique misclassifies only 3 malicious samples. This is significant because if only one malicious sample is misclassified, it causes severe damage or loss when it compromises the host. When looking at FPR, we also observe that our technique achieves the lowest FPR among the examined techniques. While a high FPR does not harm the host nor cause security risks, it affects the experience of end-users as they have to frequently whitelist the misclassified samples and cause business disruption as a false positive may trigger the entire security system to investigate. This proves the effectiveness of the developed feature vector in classifying malware.

2) Performance comparison among deep learning and machine learning models: In this section, we provide a performance comparison when using different machine learning and deep learning models on the developed feature vector. Besides ResNet1D, XGBoost, LightGBM, and MLP presented in the previous section, we perform additional experiments with many other models such as K-nearest neighbors (KNN), CatBoost, Random Forest (RF), and Logistic Regression. We note that, for these conventional machine learning algorithms, we use the default parameter setting implemented in the `scikit-learn` library[3] and feed our data for training and testing. The output of these algorithms for a given sample is the probability of the sample being malicious or not. We take the default threshold (0.5) whereas all samples that have a probability equal to or greater than the threshold are considered malicious.

In Table V, we present the performance of all the models in the descending order of detection accuracy. The experimental results show an interesting performance behavior. Generally, tree-based algorithms perform the best, followed by 1D-CNN models (i.e., ResNet1D and MLP), EfficientNetB0 (which is a 2D-CNN model) and the worst performance is Logistic Regression. We note that tree-based algorithms and ResNet1D are better than the existing techniques in various performance metrics. The reason for the performance behavior between tree-based algorithms and deep learning could be that the tree-based algorithms are very good for binary classification problems such as malware detection, whereas deep learning models exhibit their superior performance with multi-class classification problems such as face recognition or malware family classification problems. Among tree-based algorithms, gradient-boosting techniques show the best performance. This behavior aligns well with state-of-the-art research [21].

V. Conclusion

In this work, we developed a novel feature extraction method from the API calls provided in dynamic analysis re-

[3]scikit-learn: https://scikit-learn.org/stable/

TABLE IV: Performance comparison with existing methods (in percentage)

Models	Accuracy	Recall	AUC-ROC	Precision	FPR	FNR
Zhang *et al.* (LSTM) [5]	96.44 ± 0.45	95.14 ± 0.84	99.35 ± 0.17	97.66 ± 0.55	2.26 ± 0.55	4.86 ± 0.84
Rabadi *et al.* (XGBoost) [8]	96.31 ± 0.16	96.35 ± 0.42	99.42 ± 0.11	96.25 ± 0.34	3.73 ± 0.36	3.65 ± 0.42
Our Proposal (LightGBM)	$\mathbf{97.42 \pm 0.35}$	$\mathbf{97.05 \pm 0.54}$	$\mathbf{99.57 \pm 0.09}$	$\mathbf{97.75 \pm 0.27}$	$\mathbf{2.21 \pm 0.35}$	$\mathbf{2.95 \pm 0.54}$

TABLE V: Comparison with other machine learning/deep learning models (in percentage)

Models	Accuracy	Recall	AUC-ROC	Precision	FPR	FNR
LightGBM	$\mathbf{97.42 \pm 0.35}$	97.05 ± 0.54	$\mathbf{99.57 \pm 0.09}$	$\mathbf{97.75 \pm 0.27}$	$\mathbf{2.21 \pm 0.35}$	2.95 ± 0.54
XGBoost	97.11 ± 0.38	97.23 ± 0.59	99.56 ± 0.10	96.97 ± 0.42	3.02 ± 0.48	2.77 ± 0.59
CatBoost	96.94 ± 0.21	97.19 ± 0.47	99.56 ± 0.08	96.69 ± 0.32	3.30 ± 0.38	$\mathbf{2.21 \pm 0.47}$
RF	96.59 ± 0.26	$\mathbf{97.38 \pm 0.36}$	99.47 ± 0.10	95.85 ± 0.35	4.18 ± 0.41	2.62 ± 0.36
ResNet1D	96.55 ± 0.24	96.17 ± 0.30	98.42 ± 0.42	96.88 ± 0.36	3.07 ± 0.24	3.83 ± 0.30
MLP	95.42 ± 0.34	94.80 ± 1.32	98.43 ± 0.32	95.97 ± 0.76	3.97 ± 0.34	5.20 ± 1.32
KNN	94.89 ± 0.15	95.16 ± 0.22	98.07 ± 0.14	94.61 ± 3.01	5.37 ± 0.36	4.86 ± 0.22
EfficientNetB0	73.76 ± 1.13	51.55 ± 2.56	6.85 ∓ 2.31	92.37 ± 0.92	4.22 ± 0.53	48.45 ± 2.56
Logistic Regression	58.47 ± 0.78	92.90 ± 1.26	87.93 ± 0.40	54.90 ± 0.46	75.7 ± 1.20	7.1 ± 1.26

ports. New, important statistics that were previously neglected, such as return values and global statistics, are implemented in this feature vector along with the API name and its arguments. The fixed-size and the information-rich feature vector is generalized and can be used by any machine learning or deep learning models for malware detection without any dependence on the training dataset. On one hand, we used conventional machine learning algorithms and adopted existing deep learning models. On the other hand, we developed additional deep learning models (e.g., ResNet1D) to demonstrate the effectiveness of the developed feature vector. Extensive experiments show that our models outperform state-of-the-art models in various metrics, such as accuracy, precision, recall, and especially false negative rate, therefore proving the effectiveness of the proposed feature vector.

REFERENCES

[1] Kaspersky, "What is wannacry ransomware?" Feb 2022. [Online]. Available: https://shorturl.at/acEOT

[2] C. BasuMallick, "What is malware analysis? definition, types, stages, and best practices," *Spiceworks 1*, Aug 2021.

[3] A. Moser, C. Kruegel, and E. Kirda, "Limits of Static Analysis for Malware Detection," in *Proc. ACSAC'07*, 2007, pp. 421–430.

[4] M. Fazlali, P. Khodamoradi, F. Mardukhi, M. Nosrati, and M. M. Dehshibi, "Metamorphic malware detection using opcode frequency rate and decision tree," *Intl J. Info. Secur. and Pri.*, Jul. 2016.

[5] Z. Zhang, P. Qi, and W. Wang, "Dynamic malware analysis with feature engineering and feature learning," in *Proceedings of The Thirty-Fourth AAAI Conference on Artificial Intelligence*, New York, USA, 2020.

[6] M. V. Ngo, T. Truong-Huu, D. Rabadi, J. Y. Loo, and S. G. Teo, "Fast and efficient malware detection with joint static and dynamic features through transfer learning," in *21st International Conference on Applied Cryptography and Network Security*, Kyoto, Japan, June 2023.

[7] S. P. Kadiyala, A. Kartheek, and T. Truong-Huu, "Program Behavior Analysis and Clustering using Performance Counters," in *2020 Workshop on DYnamic and Novel Advances in Machine Learning and Intelligent Cyber Security (DYNAMICS'20)*, Virtual Event, Dec. 2020.

[8] D. Rabadi and S. G. Teo, "Advanced Windows Methods on Malware Detection and Classification," in *Proc. ACSAC'20*, Austin, USA, 2020.

[9] M. Rhode, P. Burnap, and K. Jones, "Early-stage malware prediction using recurrent neural networks," *Computers & Security*, vol. 77, 2018.

[10] W. Han, J. Xue, Y. Wang, L. Huang, Z. Kong, and L. Mao, "MalDAE: Detecting and explaining malware based on correlation and fusion of static and dynamic characteristics," *Computers & Security*, vol. 83, 2019.

[11] W. L. Tan and T. Truong-Huu, "Enhancing Robustness of Malware Detection using Synthetically-Adversarial Samples," in *2020 IEEE Global Communications Conference*, Taipei, Taiwan, Dec. 2020.

[12] S. Shiva Darshanm and C. Jaidhar, "Windows Malware Detector Using Convolutional Neural Network Based on Visualization Images," *IEEE Trans. Emerg. Topics Comput.*, vol. 9, no. 2, 2021.

[13] B. Ndibanje, K. H. Kim, Y. J. Kang, H. H. Kim, T. Y. Kim, and H. J. Lee, "Cross-method-based analysis and classification of malicious behavior by API calls extraction," *Applied Sciences*, vol. 9, no. 2, p. 239, 2019.

[14] X. Huang, L. Ma, W. Yang, and Y. Zhong, "A method for Windows malware detection based on deep learning," *Journal of Signal Processing Systems*, vol. 93, no. 2, pp. 265–273, 2021.

[15] X. Chen, Z. Hao, L. Li, L. Cui, Y. Zhu, Z. Ding, and Y. Liu, "Cruparamer: Learning on parameter-augmented api sequences for malware detection," *IEEE Trans. Inf. Forensics Secur*, vol. 17, pp. 788–803, 2022.

[16] Z. Salehi, A. Sami, and M. Ghiasi, "MAAR: Robust features to detect malicious activity based on API calls, their arguments and return values," *Engineering Applications of Artificial Intelligence*, vol. 59, 2017.

[17] J. Zhang, Z. Gu, J. Jang, D. Kirat, M. P. Stoecklin, X. Shu, and H. Huang, "Scarecrow: Deactivating Evasive Malware via Its Own Evasive Logic," *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 76–87, 2020.

[18] T. K. Tran and H. Sato, "NLP-based approaches for malware classification from API sequences," in *2017 21st Asia Pacific Symposium on Intelligent and Evolutionary Systems (IES)*, 2017, pp. 101–105.

[19] A. Damodaran, F. Di Troia, C. Visaggio, T. Austin, and M. Stamp, "A comparison of static, dynamic, and hybrid analysis for malware detection," in *J. Comput. Virol. Hacking Tech.*, vol. 13, no. 1, 2017.

[20] K. Weinberger, A. Dasgupta, J. Attenberg, J. Langford, and A. Smola, "Feature Hashing for Large Scale Multitask Learning," 2009. [Online]. Available: https://arxiv.org/abs/0902.2206

[21] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "LightGBM: A Highly Efficient Gradient Boosting Decision Tree," in *Proc. NeurIPS 2017*, Long Beach, California, USA, 2017.

[22] S. Bernard, L. Heutte, and S. Adam, "Random Forest Classifiers: A Survey and Future Research Directions," *Int. J. Adv. Comput.*, 2013.

[23] T. Chen and C. Guestrin, "XGBoost: A Scalable Tree Boosting System," in *Proc. ACM SIGKDD 2016*, San Francisco, California, USA, 2016.

[24] F. O. Catak, A. F. Yazı, O. Elezaj, and J. Ahmed, "Deep learning based Sequential model for malware analysis using Windows exe API Calls," *PeerJ Comput Sci.*, Jul. 2020.

[25] R. Islam, R. Tian, L. M. Batten, and S. Versteeg, "Classification of malware based on integrated static and dynamic features," *J. Netw. Comput. Appl.*, vol. 36, no. 2, 2013.

[26] J. Zhang, Z. Qin, H. Yin, L. Ou, and K. Zhang, "A feature-hybrid malware variants detection using CNN based opcode embedding and BPNN based API embedding," *Computers & Security*, vol. 84, 2019.

[27] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE CVPR 2016*, 2016, pp. 770–778.

[28] M. Tan and Q. V. Le, "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks," *CoRR*, vol. abs/1905.11946, 2019.