Advanced Windows Methods on Malware Detection and Classification

Dima Rabadi
Institute for Infocomm Research, Singapore
dimadsr@i2r.a-star.edu.sg

Sin G. Teo
Institute for Infocomm Research, Singapore
teosg@i2r.a-star.edu.sg

ABSTRACT

Application Programming Interfaces (APIs) are still considered the standard accessible data source and core wok of the most widely adopted malware detection and classification techniques. API-based malware detectors highly rely on measuring API's statistical features, such as calculating the frequency counter of calling specific API calls or finding their malicious sequence pattern (i.e., signature-based detectors). Using simple hooking tools, malware authors would help in failing such detectors by interrupting the sequence and shuffling the API calls or deleting/inserting irrelevant calls (i.e., changing the frequency counter). Moreover, relying on API calls (e.g., function names) alone without taking into account their function parameters is insufficient to understand the purpose of the program. For example, the same API call (e.g., writing on a file) would act in two ways if two different arguments are passed (e.g., writing on a system versus user file). However, because of the heterogeneous nature of API arguments, most of the available API-based malicious behavior detectors would consider only the API calls without taking into account their argument information (e.g., function parameters). Alternatively, other detectors try considering the API arguments in their techniques, but they acquire having proficient knowledge about the API arguments or powerful processors to extract them. Such requirements demand a prohibitive cost and complex operations to deal with the arguments. To overcome the above limitations, with the help of machine learning and without any expert knowledge of the arguments, we propose a light-weight API-based dynamic feature extraction technique, and we use it to implement a malware detection and type classification approach. To evaluate our approach, we use reasonable datasets of 7774 benign and 7105 malicious samples belonging to ten distinct malware types. Experimental results show that our type classification module could achieve an accuracy of 98.0253 %, where our malware detection module could reach an accuracy of over 99.8992 %, and outperforms many state-of-the-art API-based malware detectors.

CCS CONCEPTS

• **Security and privacy → Malware and its mitigation**; • **Computing methodologies → Machine learning**; **Feature selection**.

ACSAC 2020, December 7–11, 2020, Austin, USA

ACM ISBN 978-1-4503-8858-0/20/12…$15.00
https://doi.org/10.1145/3427228.3427242

KEYWORDS

API calls; feature generation; malicious behavior analysis; dynamic analysis; anomaly-based detectors; malware detection; malware type classification; machine learning

ACM Reference Format:
Dima Rabadi and Sin G. Teo. 2020. Advanced Windows Methods on Malware Detection and Classification. In *Annual Computer Security Applications Conference (ACSAC 2020), December 7–11, 2020, Austin, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3427228.3427242

1 INTRODUCTION

Despite the diversity of the available anti-virus vendors and the significant progress in malware detection research, it has been noticed that malware detection has increased year-over-year by only one percent [26]. There is a competition between a pair of almost an equal effort from two entities: 1. malware authors (cyberattackers) generating and spreading new variants of malware samples, and 2. researchers meanwhile detecting and removing them. Such competition, together with the interconnected nature of the Internet infrastructure, helped in continuously creating sophisticated malware over the years. Therefore, malware has indeed become much harsher and harder to be detected, and it is considered the most significant threat on the Internet. Thus, malware detection becomes a priority in every aware security system. The term malware represents any malicious program, software, or code, or harmful script that achieves the adversarial aim of the attacker. Once the malware is detected, the presence of its aim to harm the computer would be predicted. Malware detection has mainly two broad categories, namely static and dynamic analysis, respectively. The static analysis investigates the structure of malware's binary code without executing it, while the dynamic seeks its behavior during the execution. It has been observed that static analysis is unprotected against unknown and zero-day attacks because it has a deficiency in dealing with polymorphic malware, compression, packing, and obfuscation techniques.

Several types of research [7, 15, 24] have proven that dynamic analysis is more resilient to the techniques mentioned above and gives more reliable detection performance than the static. There are two strategies to perform dynamic analysis. In the first strategy, a predefined system state is determined at a particular point before executing the malware. Then, system states and modifications before and after that predefined point are examined to detect the malware effect on the system. In the second strategy, the malware sample is executed in an isolated and well-monitored environment, such as the sandboxing. Then, the malware behavior and its effect on the system are observed. Both strategies examine various extracted information from the system while executing the malware. Such as the registry keys modification, the accessed/modified and

dropped files, newly created and accessed processes (i.e., Application Programming Interfaces, APIs), and kernel requested services, to name a few. However, using extracted features from API calls to understand and detect the malware behavior shows promising performance using both static and dynamic analyses.

API is capable of holding enough information about the program and its behavior as it provides access to the essential resources that are available to the kernel system. API has two main parts, the function name and parameters (arguments). Function name is easily extracted as it is represented by a string that belongs to different categories (i.e., security, system services, and networking [32]). However, function parameters are very complex and belong to various types (e.g., integers, strings, and address pointers). Consequently, most of the available API-based malware detection approaches ignore the API arguments in their feature extraction techniques [14, 16, 25, 36]. API-based malware detection has received significant attention from researchers [22, 24, 35, 38, 40, 41, 54]. In literature, API-based malware detection has made its advances during three distinct stages. (1) Using static analysis to create static-signatures (byte strings) out of the extracted sequence of API calls [45]. (2) Static analysis has then evolved from using only the API signatures to also calculating various API-based statistical properties, such as the frequency counter of calling specific API calls. Then, both frequency counter and signatures are used together to detect the malware [1, 51]. However, techniques based on API signatures and statistical properties are non-resilient to malware authors who would insert redundant API calls, thus, altering the frequency counter. (3) As a result, researchers expand the API-based detection topic by moving their attention to more efficient techniques based on dynamic analysis. For example, in [24], common patterns from the API call sequences are extracted using the Longest Common Subsequence (LCS) alignment. Similarly, in [38], searching algorithms and sequence alignment using n-gram are used to find malicious API patterns. Authors of [3] study the contextual relationships between the API calls to distinguish benign from malicious behavior. They group the API calls with similar categories into clusters, then based on pattern recognition techniques (i.e., Markov Chain), the malicious sequences of API clusters are recognized from the benign. Such work is highly dependent on the sequence (order) of API calls and ignores the API arguments. However, detection techniques relying on API sequence pattern recognition can be evaded by inserting irrelevant calls or shuffling the API sequence using any off-the-shelf hooking tools. Therefore, malware authors can easily mislead both static and dynamic analyses that rely on calculating the statistical properties or recognizing the malicious sequence pattern of the API calls.

Recently, extracted features from API calls are used in various machine learning techniques such as Random Forest (RF) [44] and Sequential Minimal Optimization (SMO) [52] to implement malware detection frameworks. Such machine learning detection techniques have achieved outstanding performance and high accuracy detection rates. However, most of the available machine learning techniques try to gather domain knowledge about the arguments which requires complex operations to process them, or completely ignore the API arguments in the feature extraction operations, due to the complexity and the heterogeneous nature of the arguments. Ignoring the API arguments and relying on the API function names alone is insufficient to understand the purpose of the program. For example, the same API call (e.g., writing on a file) would act in two ways if different arguments (e.g., writing on the system versus user file) are passed to the call. As a result, other researches have applied deep learning models (e.g., Convolutional or Recurrent Neural Network, CNN, and RNN, respectively), for malicious behavior detection [21, 36, 56]. Expensive and powerful computers (e.g., a Graphics Processing Unit, GPU) would be needed to run such models.

Like legitimate Windows programs, malware has developed over the years and starts getting equipped with different functionalities and harmful goals. After detecting the malware, the next question is how to remove it from the system (e.g., malware mitigation). In malware analysis, the simple way to mitigate the malware is to know its family or type. In academia, there is still inconsistency in dealing with malware type and family concepts. As described in the well-known malware vendors (e.g., Microsoft, Malwarebytes, Symantec) and [4, 13], the malware family describes all malware samples that share the same specific functionality, or their payloads have similar source code as the origin. They behave similarly by using the same methods to achieve their goals. In contrast, a malware type describes the sample's general malicious behavior, where malware samples with the same type share a general functionality (e.g., same malicious goal) by having similar features and behavior. The scanned malware sample's detection name given by the anti-virus vendors and the multi-scanning services like VirusTotal is usually considered a malware type. Thus, as our malware labels are identified based on the information given by Malwarebytes [28] engine in VirusTotal, we only focus on malware type classification in this paper. In order to get a post malware response and well-defined defense, malware must not only be detected but also categorized based on its functionality into its type. Understanding the evolvement of different malware types over the years helps in analyzing and extracting salient features, which are then used to train good malware detection models. On the other hand, knowing how different types of malware spread is vital to containing and removing them from the system.

In this paper, we explore a new low-cost malware detection and type classification approach by studying the API calls together with their arguments. An important research question that we seek to answer is whether a low-cost API feature extraction without any domain knowledge of the arguments is sufficient to detect the malware and classify it into its type. To answer this question, we propose two methods using the extracted API arguments. In the first method, all arguments of each API call are extracted as one feature, whereas in the second method, each argument of each API call is extracted as one feature. To the best of our knowledge, no research has studied the feature extraction technique presented in the second method. We conduct extensive measurements on ten different malware types using 7774 and 7105, benign and malicious samples, respectively. Our malware detection gives an accuracy of over 99.8992 %, and outperforms the state-of-the-art API-based dynamic analysis methods. We make the following main contributions in this paper:

- We run around 14879 samples and extract their sequence of API calls and arguments.

- We propose multiple processes to study and generalize the heterogeneous API arguments without the need for explicit expertly in the domain.
- Based on the proposed processes, we design two feature extraction and generation methods (Method 1 and Method 2) without the need for any expert domain knowledge of the arguments. Method 1 uses the entire list of arguments of each API call and considers them as one feature (e.g., number of features equals the number of API calls). Method 2 generalizes each argument of each API call separately (e.g., number of features equals the number of API arguments).
- We use Method 1 and Method 2 to implement five machine learning models for both malware detection and type classification.
- Unlike the API-based malware detection methods presented in the state-of-the-art, our approach is resilient against sequence interruption, as we do not rely on the sequence (order) or the statistical features of the API calls.
- We conduct extensive experiments to evaluate the efficacy of our proposed malware detection and type classification methods. We verify by using real malware samples that API arguments are capable of detecting malware and classifying it into its type. Moreover, our methods outperform the state-of-the-art with an accuracy of over 99.8992 %.
- The experiments and the results of this paper are reproducible; we release the source code and the hash values of the used malware samples [1]. The benign and malware executable files can be shared upon direct request from authors via email.

The rest of the paper is organized as follows. Section 2 reviews malware detection and type classification related literature. Section 3 presents the working principles of the proposed approach, feature extraction, and generation. The experiments evaluation and our benign and malicious datasets are detailed in Section 4. A comparison between our proposed methods and state-of-the-art API-based malware detection techniques is shown in Section 5. Section 6 further discusses the limitations as well as the future work of this paper. Section 7 concludes.

2 RELATED WORK

In this section, we review the recent API-based malware detection and classification methods from machine learning and feature extraction and generation perspectives.

2.1 Malware Detection

A significant amount of research on static and dynamic API-based feature extraction techniques for malware detection and type classification has been conducted recently. Finding the Cosine similarity between two portable executable (PE) files [47], distinguishing malicious API sequence patterns, and calculating the frequency of calling specific API calls, to name a few, are all examples on static API-based feature extraction techniques. In [45, 54], Veeramani *et al.* and Sami *et al.* propose two respective frameworks to analyze and categorize PE files based on their relevant static API call features. Despite their high detection accuracy, it has been proven that static features are not precisely secure, as such techniques can be easily cheated by applying obfuscation tools [33].

Consequently, researchers are currently focusing on dynamic API-based feature extraction techniques [2, 24, 38, 51] as being more promising than the static. In [11], the Multiple Sequence Alignment (MSA) algorithm is used to determine the representative API pattern from the sequence of API calls. Their work can be evaded in two ways. First, by increasing the length of API calls sequence as MSA is not practical for a sequence longer than a predefined threshold. Second, by shuffling the API sequence and/or inserting irrelevant calls; thus, modifying the pattern. Similarly, in [24], Longest Common Subsequence (LCS) alignment is used to extract the representative patterns from the API calls sequence. Further, in [38], searching algorithms and sequence alignment using n-gram are used to find malicious API patterns. As previously stated, such techniques can be evaded by simply altering the pattern using simple hooking techniques.

In [2], Alazab *et al.* implement online and offline modules to extract both the spatial statistical features from the API arguments plus the temporal statistical features from the sequence of API calls. Such statistical features include mean and variance values and pointer addresses. Their modules have achieved a high detection accuracy of 96.3 %. However, malware authors can easily modify their temporal features by faking the API calls sequence. In [17], Gupta *et al.* use the Microsoft Detours library to hook user-level Windows API calls and correlate their functionalities into 26 categories based on their purposes. In [51], the frequency counters of both the API calls and their arguments are calculated, then WEKA library [18] is used to classify the samples into benign and malicious classes. Such frequency-based detection tools can be easily evaded by adding more redundant API calls and modifying the frequency counter value.

Instead of limiting the research to only investigating the malicious programs behavior, Lanzi *et al.* focus on studying the behavior of benign programs to implement n-gram and API sequence-based malware detection model [29]. They define the global behavior of benign programs and how generally they interact with the environment, by studying the interactions between benign programs and the operating system. The sequence of system calls[2] is extracted while running various normal activities on ten different computers. Once their detection model observes any new and unseen system call sequences, it either raises a false alarm or drops the malicious program that contains such sequences. Their detection model has achieved an accuracy of over 90 % with a low false-positive rate.

Other researchers suggest using dynamic and static analyses together (hybrid analysis) for more effective malware detection techniques. For instance, in [35], Bruce *et al.* propose a model to extract dynamic and static features from the sequence of API calls. Similarity-based machine learning algorithms, such as LCS, Minkowski distance, and Cosine similarity, are then trained using these features to classify the samples into benign and malicious classes. Similarly, authors of [20], propose a malware detection framework, MalDAE, that correlates dynamic and static features

[1] Click this link to download the source code and the hash values of the used malware samples.

[2] The system calls are system-level API calls which request services from the kernel operating system. API calls could be a user or system-level calls.

from the sequence of API calls. Experimental results of both works have shown that although the syntax of the dynamic and static API call features is different, there is a semantic mapping and clear relation between them. Nevertheless, the hybrid sequence feature extraction in both works ignore the API arguments and rely only on the sequence of the API calls. Thus, malware authors can easily modify such feature extraction models by interrupting the sequence.

2.2 Malware Classification

Many researchers have also used API call features to classify malware into its type. For example, in [10], Cheng *et al.* use information retrieval theory and TF-IDF weighting to generate dynamic features using system calls and their arguments. Similarly, in [42], Rieck *et al.* implement a malware classification framework, MIST, to classify malware samples based on their similar behavior. The MIST framework has two levels; the first level uses prioritized system calls alone, where the second level uses both the prioritized systems calls and their arguments. In [5], Boukhouta *et al.* propose malicious network traffic detection and classification methods. They execute each benign and malicious sample for about three minutes in an isolated sandbox. Then, deep packet inspection and flow header features are extracted from these samples. Such features include network traffic information like the number of forwarding and backward packets with their respective sizes. Then, the features are fed into different machine learning algorithms (e.g., Boosted J48, J48, Naive Bayes, and Support Vector Machine (SVM) learning) and used to train multiple detection models to distinguish the malicious from benign network traffic. Subsequently, a total of 45 features that include the total number of packets and the mean and median of packets inter-arrival time are extracted and used to train a Hidden Markov algorithm to classify the detected malware samples into their respective types.

In [34], Nair *et al.* propose an API-based malware classification method, MEDUSA, that uses the frequency counters of calling specific API calls to implement a metamorphic engine classification. They first create a signature vector out of each metamorphic type using a statistical measurement (i.e., average frequencies of selected API calls on a given malware type). However, as stated before, frequency-based classification tools can be easily evaded by adding more redundant API calls and thus modifying the value of the frequency counter.

Other researchers suggest using API call features together with features of other resources, such as the Domain Name Server (DNS), resource consumption pattern, and network traffic data, to detect and classify malware samples into their types. For instance, authors of [20, 43, 52] use feature extracted from the frequency counter of calling the API calls, together with the features of the DNS requests and the accessed files to train their respective malware classification models. Such works combine features from different resources to perform the same malware detection or classification techniques. However, authors of [20, 52] use the API without arguments, where authors of [43] take the API argument frequency-based features into consideration. However, the frequency-based tool can be easily evaded by adding more redundant API calls and modifying the frequency counter value. Authors of [39] propose an LCS-based method that uses features of both the API calls and the binary files to classify malware samples into their types. The proposed LCS method is used to measure the similarity between the malware samples. In [6], Canali *et al.* propose different API-based feature extraction models such as n-gram sequences and bags-of-words to classify the malware into its respective type. However, the proposed models have performed poorly due to some configuration issues; for example, the distance concept between the API calls is not clearly defined in the paper. In [37], Pektas *et al.* apply similar approaches like those mentioned in [39] to classify a specific malware type (i.e., ransomware). To improve the accuracy of their multi-class malware type classification model, they use additional behavioral features (i.e., registry keys, files, and mutex artifacts) besides using the same features from [39].

In contrast, we propose a malware detection and type classification approach based on extracted dynamic features from both the API calls and their arguments. Our proposed extraction methods can efficiently and effectively extract the dynamic features that are used to train malware detection and type classification approach using different machine learning algorithms. Detecting malicious behavior based on both the API calls and arguments has been slightly studied before [10, 29, 44]. Lanzi *et al.* in [29] focus on characterizing the general interactions between the benign programs and the operating system (OS) by studying the accessed OS resources (e.g., registry keys and files) while executing the benign programs. Furthermore, [10, 44] highly depend on either the order (sequence) or the frequency of API calls. However, our approach no longer depends on the frequency or order of API calls. As a result, it can be resilient against malware mutation and obfuscation techniques (e.g., reordering and/or repeating specific API calls many times). A comparison between our approach and the aforementioned API-based malicious detection approaches will be discussed in Section 5.

3 METHODOLOGY

3.1 Overview

This section presents the working principles of the proposed malware detection and type classification approach. The basic operations of our approach are illustrated in Figure 1. As shown in the figure, it has four main components:

(1) Behavior monitoring: the behavior of the malicious and benign samples is monitored while executing them in an isolated virtual machine using Cuckoo Sandbox[3]. In this paper, Cuckoo's host, where the samples are stored, is installed with Ubuntu 16.04 LTS. Cuckoo's guest, where the samples are executed, is installed with Windows 7 64-bit and several daily-use software. After executing each sample in Cuckoo's guest, the generated logs that hold the sequence of API calls and their arguments of each sample will be saved in a behavioral analysis report (e.g., report.json) and stored on Cuckoo's host.
(2) Feature extraction and generation: this component aims to extract API-based features and prepare them for the next step (e.g., machine learning). The list of API calls, together with

[3] https://cuckoosandbox.org/

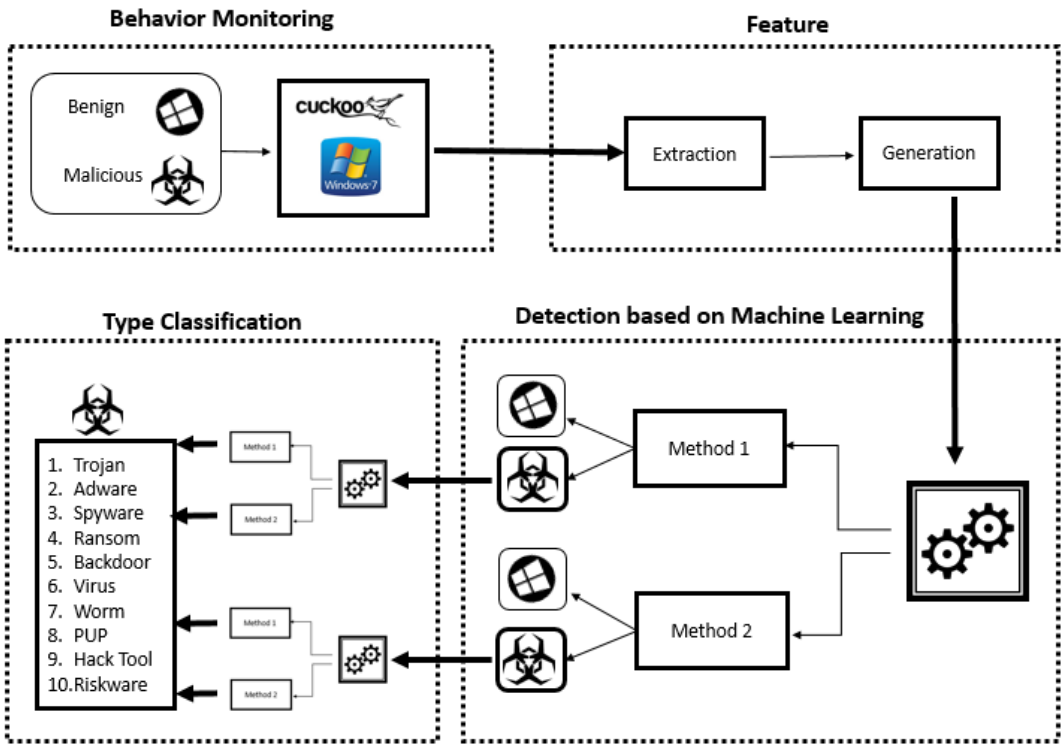

Figure 1: Overview of the proposed malware detection and type classification approach.

their arguments, are extracted from each sample's behavioral analysis report. Extracting the API calls and argument values from the report is pretty straightforward. However, analyzing the heterogeneous API arguments (e.g., different types such as hex format, integer, string, pointers, etc.) is a non-trivial task that requires some expert knowledge of the API arguments. And that is the reason behind ignoring the API arguments in most of the available machine learning malware detection tools.

The API-based features are extracted using two methods. The first method (Method 1) extracts each API call and its arguments as one token (feature), whereas the second (Method 2) deals with each argument element of each API call separately as one token. The API arguments are very heterogeneous and highly dependent on the operating system resources. Therefore, we propose dynamic feature generation techniques to process and generalize the arguments and to make them easier to be read and processed later by the machine learning algorithms in the third component. Hashing Vectorizer function [53] is then used to encode the generalized API-based dynamic features into a bit-vector (i.e., Method 1 bit-vector and Method 2 bit-vector) where the length of each bit-vector is equal to 2^{20}. This component will be demonstrated via examples in Section 3.3.

(3) Malware detection using machine learning algorithms: in this component, five different machine learning algorithms (e.g., Support Vector Machine (SVM), Gradient Boosting (XGBoost), Random Forest (RF), Decision Tree (DT), and Passive-Aggressive (PA)) are trained using the API-based dynamic features (i.e., bit-vectors from the previous step), resulting in classifying samples as benign or malicious.

(4) Malware type classification using machine learning algorithms: after detecting the malware samples in the previous step, the bit-vectors of Method 1 and Method 2 are used to train the same five machine learning algorithms (e.g., SVM, XGBoost, RF, DT, and PA), resulting in classifying the malicious samples into their respective types.

3.2 Feature Extraction and Generation (Tokenization)

This component starts by extracting the API calls and their arguments from the behavioral analysis report of each sample. Each API call has two parts; the API name (function name) and the list of its arguments. The API name is represented by a string of words, where each word starts with a capitalized letter, such as "GetFileVersionInfoSize", and could be any of the 312 API calls that are hooked by Cuckoo [46]. In addition, some of the API names end with various suffixes such as *Ex, A, W, ExA,* and *ExW*. We remove such suffixes to ensure that the extracted features are resilient against the conflict of using multiple versions of the same API call. For example, the two API calls *FindFirstFileExW* and *FindFirstFileExA* will be merged under the same API call, *FindFirstFile*. Mainly these

two API calls perform the same functionality where the difference is in the format of their arguments. For instance, the former receives its arguments in Unicode format (*ExW*), while the latter receives its arguments in ANSI coded format (*ExA*).

On the other hand, API arguments are very heterogeneous, so most researchers [14, 25, 36] neglect the API arguments and only consider the API names. However, many essential information is lost when API arguments are ignored. For example, operations such as writing/reading to/from a specific file are considered naive operations if their arguments are ignored (e.g., filenames). Nevertheless, such operations can be critical if the filename argument is a system file. Thus, always the arguments should be considered while studying the behavior of any program operations.

Now, the goal is to study and generalize the heterogeneous API arguments without the need for any manual effort or explicit expertly in the domain. To do that, we propose the following automatic processes. The Algorithms in Appendix A give the pseudocode of each of the proposed processes.

(a) API arguments with integer type are converted into their logarithmic bins. If the argument value represents an integer larger than 0, a prefix *numB* plus the logarithmic bin of the value is added to the feature. Furthermore, if the argument value is a negative number, a prefix *neg_* is added before the logarithmic bin in the feature. To distinguish 0 from the number in the range of $(0-10)$, 0 is tagged as *num*0 instead of *numB*0. This process is implemented in Algorithm 2. Moreover, arguments with concrete values will not be changed in the feature. For example, *allocation_type* argument in *NtAllocateVirtualMemory* API call has a concrete number equals 8192 that represents a *MEM_RESERVE*. Such constant value will not be changed in this process. This is implemented by Lines 20 to 23 in Algorithm 1.

(b) If the argument defines a directory path (URL), then it is checked whether it contains a System32 in its path to be categorized as *sys_dir*; otherwise, it will be added to the *other_dir* category. This process is implemented in Algorithm 5.

(c) The extension of the file is checked, whether it is one of the most popular Windows file types that are usually used by malware authors [49]. This is implemented in Algorithm 4. Similarly, if the argument is a registry key (e.g., *regkey* such as *BootExecute*, *Winlogon*, *Run*, and *RunOnce keys*, etc.), then its value is checked, whether it is one of the most common registry keys that is exploited by malware. This is implemented in Algorithm 3.

After applying the previous processes, the API-based features will be presented as a set of unique strings in the form of an API function name and the values of its argument. Algorithm 1 in Appendix A gives the pseudocode of the whole proposed API-based feature extraction and generation. The two methods will prepare the features in two different formats. In the first method, each API call and the list of its arguments are presented as one feature. Thus, we have i features, where i is the number of API calls. Whereas in the second method, each API call and each element of its arguments are considered as one feature. Thus, we have $\sum_{z=1}^{i} j_z$ features, where j_z defines the number of arguments of API_z, where $z \in (0, i)$. Consequently, Method 1 and Method 2 features are constructed using the following formulas:

- **Method 1**: 'API_0: ARG_0; ARG_1...; ARG_{j_0}', 'API_1: ARG_0; ARG_1...; ARG_{j_1}', ..., 'API_i: ARG_0; ARG_1; ...; ARG_{j_i}'
- **Method 2**: 'API_0: ARG_0', 'API_0: ARG_1', ..., 'API_0: ARG_{j_0}', 'API_1: ARG_0', 'API_1: ARG_1', ..., 'API_1: ARG_{j_1}', ..., 'API_i: ARG_0', 'API_i: ARG_1', ..., 'API_i: ARG_{j_i}'

After feature extraction and generation, Hashing Vectorizer function [53] is then used to encode the generalized API-based dynamic features into fixed-length bit-vector. Each method's dynamic features will be encoded into its bit-vector, Method 1 bit-vector and Method 2 bit-vector, where the length of each bit-vector equals 2^{20}.

3.3 Practical Examples

This section shows numerical examples of our proposed feature extraction and generation methods and the Hashing Vectorizer function.

3.3.1 Example - Feature Extraction and Generation: Listing 1 shows a sequence of two API calls (*NtQueryValueKey* and *NtAllocateVirtualMemory*) and their arguments in a JSON format, which are extracted from one of the samples' analysis report. As shown, *reg_type* and *information_class* arguments have values of 0 and 1, respectively. The feature generation processes presented before convert these values to *num*0 and *numB*0 (e.g., numB+ $\log_{10}$ 1), respectively. Table 1 shows the generated features of Method 1 and Method 2 of the two API calls and their arguments presented in Listing 1.

Listing 1: Two API calls and their arguments snippet from Cuckoo's analysis report (JSON format).

```
\\report.json
"api": "(*\bfseries NtQueryValueKey*)",
"return_value": 3221225524,
"(*\bfseries arguments*)": {
"value": " ",
"reg_type": 0,
"information_class": 1,
"regkey": "HKEY_LOCAL_MACHINE/SYSTEM/
ControlSet001/Control/en-US"},

"api": "(*\bfseries NtAllocateVirtualMemory*)",
"return_value": 0,
"(*\bfseries arguments*)": {
"process_identifier": 3336,
"region_size": 524288,
"stack_dep_bypass": 0,
"stack_pivoted": 0,
"heap_dep_bypass": 0,
"protection": 4,
"allocation_type": 8192},
```

3.3.2 Example - Hashing Vectorizer: After the dynamic features of Method 1 and Method 2 are extracted, the Hashing Vectorizer function will be applied to encode the features into a bit-vector for each method. In this example, we show the bit-vector generation of Method 2, where Method 1 is the same, but all arguments of each API call are considered as one feature. Let us assume that our dataset contains three samples (i.e., Sample 1, Sample 2, and Sample 3). Figure 2 shows the corpus, the extracted dynamic features of

Table 1: The generated features of the two API calls and their arguments presented in Listing 1.

Method	Features
Method 1	'NtQueryValueKey:0=numB0; NtQueryValueKey:1=0; NtQueryValueKey:2=other_reg; NtQueryValueKey:3=num0', 'NtAllocateVirtualMemory:0=8192; NtAllocateVirtualMemory:1=num0; NtAllocateVirtualMemory:2=numB3; NtAllocateVirtualMemory:3=numB0; NtAllocateVirtualMemory:4=numB5; NtAllocateVirtualMemory:5=num0; NtAllocateVirtualMemory:6=num0'
Method 2	'NtQueryValueKey:0=numB0', 'NtQueryValueKey:1=0', 'NtQueryValueKey:2=other_reg', 'NtQueryValueKey:3=num0', 'NtAllocateVirtualMemory:0=8192', 'NtAllocateVirtualMemory:1=num0', 'NtAllocateVirtualMemory:2=numB3', "NtAllocateVirtualMemory:3=numB0', 'NtAllocateVirtualMemory:4=numB5', 'NtAllocateVirtualMemory:5=num0', 'NtAllocateVirtualMemory:6=num0'

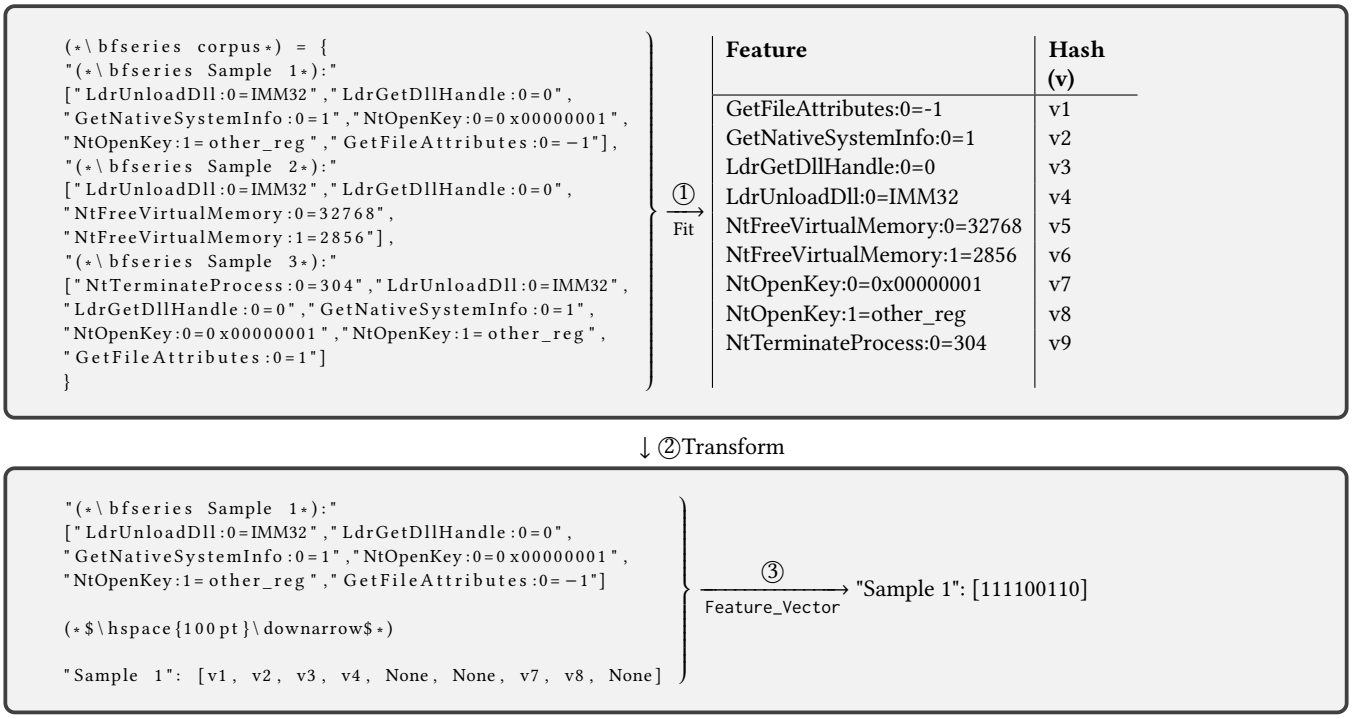

Figure 2: Hashing Vectorizer function example.

each sample using Method 2 (total of nine features). The Hashing Vectorizer function works in two steps, Fit and Transform. Fit starts by identifying all unique features in corpus and hashing each feature (i.e., the used hash function is signed 32-bit version of Murmurhash3 [53]). As a result, the hash table and feature space will be formed, as shown in the first step in Figure 2. Then, in the second step, Transform starts by hashing each feature in each sample of the corpus. Based on the hash table, which is formed by the Fit function, it matches each feature's hash value with that of the hash table. Thus, it gets the corresponding index of the matched hash in the hash table. Lastly, a value of 1 will be assigned to the index, forming the feature bit-vector. Step 3 in Figure 2 shows Sample 1 after rearranging the features in ascending order (`Feature_Vector`).

4 EXPERIMENTS

4.1 Datasets

Our datasets include 7105 and 7774, malicious and benign samples, respectively. The malicious samples are obtained from the Malshare website [27] using a daily downloading script. Each sample is then validated and archived by its date using VirusTotal [30]. Five antivirus engines need to vote that the sample is malicious to be considered in our dataset (i.e., Microsoft, Malwarebytes, Symantec, Sophos, and Trend Micro). As described before, malware samples with the same type have similar features and behavior. However, getting the malware type's ground-truth label is non-trivial, where multiple anti-virus vendors would give different detection names

Table 2: Dataset description.

Sample	Type	No. of samples	%
Malicious	Trojan	5485	36.864
	Adware	295	1.983
	Spyware	381	2.561
	Ransom	441	2.964
	Backdoor	191	1.284
	Virus	99	0.665
	Worm	152	1.022
	PUP	36	0.242
	Hack Tool	14	0.094
	Riskware	11	0.074
Benign	Windows executable	327	2.198
	DLL files	1069	7.185
	APIMDS	200	1.344
	CNET	173	1.163
	Portable applications	300	2.016
	File Hippo	43	0.289
	CYGWIN	4631	31.124
	WINDOWS 7	1031	6.929
Total		14879	-

(types) for the same scanned sample. Thus, researchers start looking into another way of labeling the malware samples. For example, in [48], Sebastián *et al.* implement AVClass, which is an open-source and automatic tool that takes a malware sample and returns its type label with a confidence factor calculated by leveraging the anti-virus labels obtained from VirusTotal and the agreement obtained across its engines. The ground-truth labeling is out of this paper scope, where for our malware dataset, the malware type of each sample is named based on the information given by Malwarebytes [28] engine in VirusTotal. The age of our malicious samples is between January 2019 and March 2020, based on VirusTotal results. The malware types and the number of samples in each type are presented in Table 2. A description of each malware type used in this paper is shown in Table 6 in Appendix C.

The benign samples are obtained from eight different sources. (1) After immediately installing a fresh version of Windows 7, we extracted the Native Windows executable, and (2) the Dynamic Link Library (DLL) files which are located in *C:\ Windows\ Systems32* directory. (3) We downloaded the APIMDS dataset [55]. (4) We used websites for free-to-try legal downloads (e.g., download.cnet.com and softpedia.com). (5) We downloaded the top 300 portable applications for Windows [19] and (6) the top 43 applications from the File Hippo Website [50]. (7) We used the benign dataset from [23], which contains two folders, CYGWIN [8] and (8) WINDOWS 7 benign samples. Both folders contain Windows executable files that are copied from the authors directly. All benign samples from the eight sources are also validated using VirusTotal. The number of samples of each benign source is presented in Table 2.

4.2 Method Evaluation

In this section, we evaluate the efficacy of our proposed methods in detecting the malware from benign samples and then classifying them into their respective types. The benign dataset is split into 6220 (80 % of the dataset) and 1554 (20 % of the dataset) samples for training and testing, respectively. Similarly, for a fair evaluation, each malware type dataset is split into training and testing datasets, which consist of 80 % and 20 % of the samples, respectively. Thus, in total, we have 5687 and 1418 malware samples for training and testing, respectively.

4.2.1 Malicious Behavior Detection. In this set of experiments, we investigate the performance of Method 1 and Method 2 in detecting malicious from benign samples. To do that, we use the 5687 and 6220, malicious and benign training datasets, respectively, to train models using five different machine learning algorithms (i.e., SVM, XGBoost, RF, DT, and PA). To avoid the overfitting, we apply 5-fold cross-validation over the malicious and benign training datasets using the above machine learning algorithms. The models are then used to test the 1418 and 1554 malicious and benign testing datasets, respectively. The following standard machine learning performance metrics are used to evaluate the performance of the proposed methods:

(1) False Positive Rate (FPR): the ratio of falsely classifying benign samples as malicious.
(2) False Negative Rate (FNR): the ratio of falsely classifying malicious samples as benign.
(3) Accuracy: how often is our method correct, which denotes the rate of correct outcomes (benign and malicious) out of the observed samples.
(4) Standard Deviation (STD): a positive or negative value represents how the array of scores of the estimator for each run of the cross-validation may differ from the mean value (Mean Accuracy).

Table 3 shows Method 1 and Method 2 performance results. As shown in the table, XGBoost achieves the best accuracy score among the five machine learning algorithms using our proposed Method 1 and Method 2, where Method 1 gets 99.8655 and Method 2 gets 99.8992. Thus, we further run the XGBoost experiments using the 10-fold cross-validation. Method 1 and Method 2 performance results are then evaluated in terms of FPR, FNR, and:

(1) Precision: the ratio of correctly predicted positive observations to the total predicted positive observation given by $\left(\frac{TP}{TP+FP}\right)$.
(2) Recall: the ratio of the correctly detected malicious samples over all malicious samples $\left(\frac{TP}{TP+FN}\right)$.
(3) F1 score: the harmonic mean of precision and recall given by $\left(2 \cdot \frac{Precision \cdot Recall}{Precision+Recall}\right)$.

Table 4 shows the results. A comparison between the performance of Method 1 and Method 2 using XGBoost and discussion on their misclassification samples will be presented in the following.

4.2.2 Methods Performance and Misclassifications. In this section, we present the performance difference of Method 1 and Method 2 using XGBoost, and under which circumstances Method 2 would outperform Method 1. As described before, the two methods serve the same purpose, where the main difference between them lies in the tokenization strategies. Method 1 considers the entire list of arguments of each API call as one token, while Method 2 considers

Table 3: The malicious behavior detection experimental results.

Machine Learning Algorithms	Method	FPR	FNR	Accuracy	5-fold cross-validation	
					Mean Accuracy	(±) STD
SVM	1	0.00192	0.00985	99.4287	96.2	0.079
	2	0.00192	0.00422	99.6976	96.0	0.106
Random Forest	1	0.00836	0.01900	98.6559	95.4	0.078
	2	0.00385	0.00633	99.4959	94.4	0.116
Passive Aggressive	1	0.00257	0.01337	99.2271	96.0	0.088
	2	0.00321	0.00422	99.6303	95.7	0.111
Decision Tree	1	0.00385	0.00492	99.5631	92.1	0.245
	2	0.01221	0.00422	99.1599	95.2	0.133
XGBoost	1	**0.00128**	**0.00140**	**99.8655**	**95.7**	**0.128**
	2	**0.00128**	**0.00070**	**99.8992**	**96.7**	**0.086**

Table 4: The malicious behavior detection - XGBoost experimental results.

Method	**Type**	**FPR**	**FNR**	**Precision**	**Recall**	**F1 Score**	**10 fold cross-validation**	
							Mean Accuracy	**(±)STD**
Method 1	Benign	0.00128	-	99.87	99.87	99.87	99.2	0.024
	Malicious	-	0.00141	99.86	99.86	99.86		
	Total	0.00128	0.00141	99.87	99.87	99.87		
Method 2	Benign	0.00128	-	99.94	99.87	99.90	99.4	0.020
	Malicious	-	0.00070	99.86	99.93	99.89		
	Total	0.00128	0.00070	99.90	99.90	99.90		

each argument of each AP call separately as one feature. If the sample has called a small number of API calls, but the number of the passed arguments for each API call is high, then Method 2 outperforms Method 1, as the large number of arguments for each call can compensate the small number of total API calls. This observation has been proven in Table 4. It clearly shows that Method 2 has slightly better performance in detecting malware than Method 1.

Next, we discuss some reasons to cause the misclassifications in our proposed methods. In Table 4, Method 1 has falsely classified a few malicious samples as benign, that causes a false negative rate of 0.00140. The main reason behind such misclassification is that the number of API calls of one of these samples is small (i.e., seven API calls). Thus, only seven features are generated using Method 1. In contrast, Method 2 could generate around 60 features out of these seven API calls and correctly classify this sample as malicious (thus, reducing the FNR to 0.00070 in Method 2). The experimental results presented above show that our approach can distinguish benign from malicious behavior with high accuracy. However, our approach, like any other detection approaches, misclassifies a small number of samples that causes the FPR as presented in Tables 3 and 4. After studying the misclassified benign samples, which caused the FPR in our approach, the number of API calls of the samples is only two, where each API call has only one argument (e.g., "__exception__:0=numB0", "NtTerminateProcess:0=numB2"). Thus, both methods have only two features. Due to such small numbers of extracted features from these samples, our approach has misclassified them as malicious; as a result, we get the small FPR.

4.2.3 Malware Type Classification. The goal of this set of experiments is to investigate the performance of Method 1 and Method 2 in classifying the malware samples into their respective types. As discussed before, our malicious samples belong to ten malware types (Table 2), the same features that are used to classify the samples into malicious and benign classes are again used to classify the malicious samples into their types. For a good validation and to make sure that the machine learning modules are trained using a fair amount of samples from each type, each malware type dataset is split into training and testing datasets, which consist of 80 % and 20 % of the samples, respectively. Thus, in total, we have 5687 malware samples for training and 1418 for testing. The training dataset is used to train models using the same five machine learning algorithms as before. From the experimental results in Figure 3, XGBoost performs as the best malware type classifier out of the five classifiers. It gives the best accuracy values (98.0253 %) for Method 1 and (97.9548 %) for Method 2.

We apply 5-fold cross-validation over the five machine learning algorithms. Results are shown in the bottom half of Figure 4. As shown in the figure, the accuracy is in the range between [91.9 % - 93.4 %]. Such decreasing in the accuracy values is due to the small number of samples in some of the types (e.g., Backdoor, Virus, Worm, PUP, Hack Tool, and Riskware). To overcome this issue, we remove individual types which have less than 200 samples in our model training. In other words, the experiments are done using the samples that belong to only the following four malware types: Trojan, Adware, Spyware, and Ransom. Again, we apply the 5-fold cross-validation over the five machine learning algorithms using

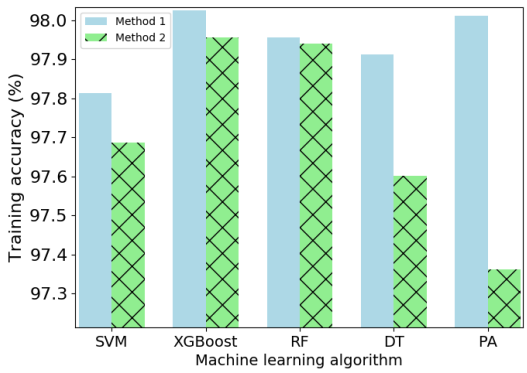

Figure 3: Training accuracy results of malware type classification.

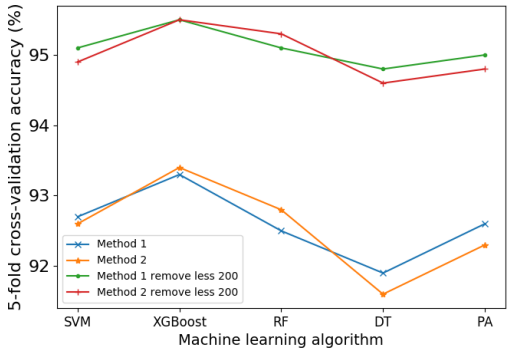

Figure 4: The 5-fold cross-validation over the five machine learning algorithms. The bottom half shows the results using the whole malicious dataset (i.e., ten malware types). The top half shows the results using only four malware types, after removing the types that have less than 200 samples.

the four malware types. The results are shown in the top half of Figure 4. We clearly see that the accuracy values have increased to the range between [94.6 % - 95.5 %]. The experiments have proven that increasing the number of the malware samples of each type can improve the performance of Method 1 and Method 2 in classifying malware samples into their types.

5 COMPARISON WITH STATE-OF-THE-ART

In this section, we compare our methods with other works that consider the API arguments in their techniques (i.e., [1, 10, 40, 42, 44, 51]). Our comparison include (i) Detection accuracy (ii) Required API information such as finding the frequency counter of calling specific API calls or recognizing the API sequence patterns, and (iii) Limitations.

The following studies have used the API arguments to establish malware detection and/or type classification models [1, 10, 40, 42, 44, 51]. As discussed in Section 2, both [44] and [10] use pattern recognition techniques to find a mutual sequence of API calls and arguments. However, the sequence (i.e., pattern) of API calls can be disrupted by deleting and/or inserting certain API calls. In contrast, [1, 40, 51] use API frequency-based malware detection techniques. In [51], the frequency metric of calling specific API calls and their arguments has been used to differentiate between benign and malware samples. In [40], Yong *et al.*, have used frequent item-sets of API calls and their arguments for malware detection. Similarly, Faraz *et al.* [1] have used statistical features of API calls and their arguments, such as the frequency, mean, and size of parameters, to detect the malicious behavior of programs. Malware authors can easily bypass the above mentioned frequency-based approaches by eliminating and/or adding API calls hence altering the frequency counter values.

Our methods are different from the studies above. (i) Our methods are resilient to malware mutation and obfuscation techniques (e.g., reordering the sequence of API calls or repeating specific API calls and/or arguments many times) because we do not rely on the sequence or the pattern of the API calls and their arguments. Instead, only the occurrence of the API calls and their arguments is considered in our methods. (ii) The statistical features such as mean, frequency, or the size of the API arguments, are not considered in our approach. (iii) Domain knowledge of the complex arguments is not required in our methods as we implement a novel feature generation functions that are used to enhance the extracted API-based features for better processing by the machine learning algorithms. (iv) None of the existing approaches have studied the possibility of using each argument element of each API call separately, as presented in Method 2. These advantages execute our approach against the scalability issue of using high dimensional feature space, which requires high memory consumption and increases the computational complexity. We summarize the comparison between our approach and the above-mentioned related studies in Table 5. Our proposed methods have outperformed the state-of-the-art methods as shown in Table 5. However, as [44] gives the highest F1 score among the six works, we use it as a baseline. Appendix D shows the evaluation between our proposed methods against the baseline [44] and [51] in terms of the recall scores.

6 LIMITATIONS AND FUTURE WORK

In this paper, we have only targeted Windows 7. We plan to explore the performance of our proposed methods on other newly released versions of Windows (i.e., Windows 10). Furthermore, Android is an API-based operating system; thus, we plan to test the performance of our methods on Android platforms. In addition, all experiments are conducted using the Cuckoo sandbox. Therefore, this paper is limited to the list of Cuckoo's hooked API calls [46]. In the future, we plan to design a run-time malware analysis and detection tool that can extract the API calls simultaneously while the program is running. Furthermore, we have tested our methods on a dataset of 14879 samples. To check the performance of our methods on more massive datasets, we are collecting daily and up-to-date malicious samples and generating daily benign samples. In future work, we plan to leverage the existing model by considering more complicated malware classification scenarios where, for

Table 5: Comparison with API arguments-based malware detection methods.

Approach	F1 (%)	Techniques	API Information	Limitations
[10]	97.7	Information retrieval theory, TF-IDF weighting	The sequence of Windows API calls and their arguments	Sequence-based[1]
[44]	97.9	Random forest, J48 decision tree, Sequential Minimal Optimization (SMO)	The sequence of API calls including their arguments and/or return value	A subset of 126 API calls, Sequence-based[1]
[42]	96.7	n-gram	System calls and arguments	System calls only
[51]	97.0	Weka library, SVM, decision table, random forest, Instance-based classifier (IB1)	Frequency counters of API calls with and without arguments	Frequency-based[2]
[40]	94.7	n-gram	Frequent item-sets of the sequence of API calls with and without their arguments	Frequency-based[2]
[1]	96.6	SMO, RIPPER, NB, J48, IB_k	Statistical features of API calls and their arguments (e.g., frequency, mean and size parameters)	Frequency-based[2]
Method 1 **Method** 2	99.87 99.90	SVM, XGBoost[3], RF, DT, PA	Method 1: API call and its entire list of arguments, Method 2: API call and each of its argument element separately	Section 6

[1] Sequence-based: malware authors can bypass sequence-based techniques by deleting and/or inserting API calls.
[2] Frequency-based: malware authors can bypass frequency-based techniques by removing and/or adding API calls hence altering the frequency counter value.
[3] The reported F1 scores of Method 1 (99.87 %) and Method 2 (99.90 %) are based on the XGBoost classifier as it achieves the best accuracy and F1 scores among the five machine learning algorithms.

example, one malware sample can simultaneously perform multiple functionalities. Therefore, multi-labeling algorithms would be required.

7 CONCLUSIONS

This paper explored a new direction to extract the API-based dynamic features by analyzing the API calls together with their list of arguments. With the help of machine learning algorithms, we designed two methods to detect Windows malware samples and classify them into their types. The first method deals with the entire list of arguments of each API call as one feature, whereas the second method deals with each argument of each API call separately as one feature. We verified the performance of the proposed methods in malware detection using reasonable datasets of 7105 malicious samples belonging to ten distinct types and 7774 benign samples. We showed that our approach outperforms other recent API arguments-based malware detection approaches in terms of accuracy, limitations, and required API information. Our experimental results showed that our malware detection approach gave an accuracy of over 99.8992 %, and outperformed the state-of-the-art. Furthermore, we used the same methods to classify the malware samples into their types. The experimental results showed that our malware classification approach gave an accuracy of over 97.9548 %.

Our approach has promise for wide adoption in API-based malicious behavior detection for Windows platforms. In particular, it can meet the demands of applications that are currently served by malware detectors that rely on extracting statistical information of the API-based dynamic features but desire better robustness against unfavorable sequence interruption attacks.

ACKNOWLEDGMENTS

The authors wish to thank our shepherd Dr. Kevin Alejandro Roundy, and the anonymous reviewers for providing valuable feedback on this work. We also thank Loo Jia Yi from the Cyber Security Department at Institute for Infocomm Research for evaluating the Machine Learning classifiers mentioned in this work.

REFERENCES

[1] Faraz Ahmed, Haider Hameed, M Zubair Shafiq, and Muddassar Farooq. 2009. Using spatio-temporal information in API calls with machine learning algorithms for malware detection. In *Proceedings of the 2nd ACM workshop on Security and artificial intelligence*. ACM, 55–62.
[2] Mamoun Alazab, Sitalakshmi Venkataraman, and Paul Watters. 2010. Towards understanding malware behaviour by the extraction of API calls. In *2010 Second Cybercrime and Trustworthy Computing Workshop*. IEEE, 52–59.
[3] Eslam Amer and Ivan Zelinka. 2020. A dynamic Windows malware detection and prediction method based on contextual understanding of API call sequence. *Computers & Security* 92 (2020), 101760.

[4] Sergii Banin and Geir Olav Dyrkolbotn. 2018. Multinomial malware classification via low-level features. *Digital Investigation* 26 (2018), S107–S117.

[5] Amine Boukhtouta, Serguei A Mokhov, Nour-Eddine Lakhdari, Mourad Debbabi, and Joey Paquet. 2016. Network malware classification comparison using DPI and flow packet headers. *Journal of Computer Virology and Hacking Techniques* 12, 2 (2016), 69–100.

[6] Davide Canali, Andrea Lanzi, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, and Engin Kirda. 2012. A quantitative study of accuracy in system call-based malware detection. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. 122–132.

[7] Silvio Cesare and Yang Xiang. 2012. *Software similarity and classification*. Springer Science & Business Media.

[8] Steve Chamberlain and Cygnus Solutions. [n.d.]. Cygwin. ([n. d.]). https://cygwin.com/.

[9] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 785–794.

[10] Julia Yu-Chin Cheng, Tzung-Shian Tsai, and Chu-Sing Yang. 2013. An information retrieval approach for malware classification based on Windows API calls. In *2013 International Conference on Machine Learning and Cybernetics*, Vol. 4. IEEE, 1678–1683.

[11] In Kyeom Cho and Eul Gyu Im. 2015. Extracting representative API patterns of malware families using multiple sequence alignments. In *Proceedings of the 2015 Conference on research in adaptive and convergent systems*. ACM, 308–313.

[12] Koby Crammer, Ofer Dekel, Joseph Keshet, Shai Shalev-Shwartz, and Yoram Singer. 2006. Online passive-aggressive algorithms. *Journal of Machine Learning Research* 7, Mar (2006), 551–585.

[13] G DATA. [n.d.]. Malware Naming Hell Part 1: Taming the mess of AV detection names. ([n. d.]). https://www.gdatasoftware.com/blog/2019/08/35146-taming-the-mess-of-av-detection-names.

[14] Omid E David and Nathan S Netanyahu. 2015. Deepsign: Deep learning for automatic malware signature generation and classification. In *2015 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1–8.

[15] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. 2012. A survey on automated dynamic malware-analysis techniques and tools. *ACM computing surveys (CSUR)* 44, 2 (2012), 6.

[16] Seoungyul Euh, Hyunjong Lee, Donghoon Kim, and Doosung Hwang. 2020. Comparative Analysis of Low-Dimensional Features and Tree-Based Ensembles for Malware Detection Systems. *IEEE Access* 8 (2020), 76796–76808.

[17] Sanchit Gupta, Harshit Sharma, and Sarvjeet Kaur. 2016. Malware characterization using windows API call sequences. In *International Conference on Security, Privacy, and Applied Cryptography Engineering*. Springer, 271–280.

[18] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. 2009. The WEKA data mining software: an update. *ACM SIGKDD explorations newsletter* 11, 1 (2009), 10–18.

[19] John T. Haller. [n.d.]. Portable Apps. ([n. d.]). https://portableapps.com/.

[20] Weijie Han, Jingfeng Xue, Yong Wang, Lu Huang, Zixiao Kong, and Limin Mao. 2019. MalDAE: Detecting and explaining malware based on correlation and fusion of static and dynamic characteristics. *Computers & Security* 83 (2019), 208–233.

[21] Xiang Huang, Li Ma, Wenyin Yang, and Yong Zhong. 2020. A Method for Windows Malware Detection Based on Deep Learning. *Journal of Signal Processing Systems* (2020), 1–9.

[22] Kazuki Iwamoto and Katsumi Wasaki. 2012. Malware classification based on extracted api sequences using static analysis. In *Proceedings of the Asian Internet Engineeering Conference*. ACM, 31–38.

[23] Arzu Gorgulu Kakisim, Mert Nar, Necmettin Carkaci, and Ibrahim Sogukpinar. 2018. Analysis and evaluation of dynamic feature-based malware detection methods. In *International Conference on Security for Information Technology and Communications*. Springer, 247–258.

[24] Youngjoon Ki, Eunjin Kim, and Huy Kang Kim. 2015. A novel approach to detect malware based on API call sequence analysis. *International Journal of Distributed Sensor Networks* 11, 6 (2015), 659101.

[25] Bojan Kolosnjaji, Apostolis Zarras, George Webster, and Claudia Eckert. 2016. Deep learning for classification of malware system call sequences. In *Australasian Joint Conference on Artificial Intelligence*. Springer, 137–149.

[26] Malwarebytes Labs. [n.d.]. 2020 State of Malware Report. ([n. d.]). https://resources.malwarebytes.com/files/2020/02/2020_State-of-Malware-Report.pdf.

[27] Malshare Labs. [n.d.]. Malshare Website. ([n. d.]). https://malshare.com/.

[28] Malwarebytes Labs. [n.d.]. Malware Types. ([n. d.]). https://www.malwarebytes.com/malware/.

[29] Andrea Lanzi, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, and Engin Kirda. 2010. Accessminer: using system-centric models for malware protection. In *Proceedings of the 17th ACM conference on Computer and communications security*. 399–412.

[30] VirusTotal malware intelligence services. [n.d.]. https://www.virustotal.com. ([n. d.]).

[31] Stephen Marsland. 2015. *Machine learning: an algorithmic perspective*. CRC press.

[32] Microsoft. [n.d.]. Overview of the Windows API. ([n. d.]). https://docs.microsoft.com/en-us/previous-versions//aa383723(v=vs.85).

[33] Andreas Moser, Christopher Kruegel, and Engin Kirda. 2007. Limits of static analysis for malware detection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. IEEE, 421–430.

[34] Vinod P Nair, Harshit Jain, Yashwant K Golecha, Manoj Singh Gaur, and Vijay Laxmi. 2010. Medusa: Metamorphic malware dynamic analysis usingsignature from api. In *Proceedings of the 3rd International Conference on Security of Information and Networks*. 263–269.

[35] Bruce Ndibanje, Ki Hwan Kim, Young Jin Kang, Hyun Ho Kim, Tae Yong Kim, and Hoon Jae Lee. 2019. Cross-Method-Based Analysis and Classification of Malicious Behavior by API Calls Extraction. *Applied Sciences* 9, 2 (2019), 239.

[36] Razvan Pascanu, Jack W Stokes, Hermineh Sanossian, Mady Marinescu, and Anil Thomas. 2015. Malware classification with recurrent networks. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 1916–1920.

[37] Abdurrahman Pektaş and Tankut Acarman. 2017. Classification of malware families based on runtime behaviors. *Journal of information security and applications* 37 (2017), 91–100.

[38] Abdurrahman Pektaş and Tankut Acarman. 2017. Malware classification based on API calls and behaviour analysis. *IET Information Security* 12, 2 (2017), 107–117.

[39] Radu S Pirscoveanu, Steven S Hansen, Thor MT Larsen, Matija Stevanovic, Jens Myrup Pedersen, and Alexandre Czech. 2015. Analysis of malware behavior: Type classification using machine learning. In *2015 International conference on cyber situational awareness, data analytics and assessment (CyberSA)*. IEEE, 1–7.

[40] Yong Qiao, Yuexiang Yang, Lin Ji, and Jie He. 2013. Analyzing malware by abstracting the frequent itemsets in API call sequences. In *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, 265–270.

[41] Chandrasekar Ravi and R Manoharan. 2012. Malware detection using windows api sequence and machine learning. *International Journal of Computer Applications* 43, 17 (2012), 12–16.

[42] Konrad Rieck, Philipp Trinius, Carsten Willems, and Thorsten Holz. 2011. Automatic analysis of malware behavior using machine learning. *Journal of Computer Security* 19, 4 (2011), 639–668.

[43] Zahra Salehi, Mahboobeh Ghiasi, and Ashkan Sami. 2012. A miner for malware detection based on API function calls and their arguments. In *The 16th CSI international symposium on artificial intelligence and signal processing (AISP 2012)*. IEEE, 563–568.

[44] Zahra Salehi, Ashkan Sami, and Mahboobe Ghiasi. 2017. MAAR: Robust features to detect malicious activity based on API calls, their arguments and return values. *Engineering Applications of Artificial Intelligence* 59 (2017), 93–102.

[45] Ashkan Sami, Babak Yadegari, Hossein Rahimi, Naser Peiravian, Sattar Hashemi, and Ali Hamze. 2010. Malware detection based on mining API calls. In *Proceedings of the 2010 ACM symposium on applied computing*. ACM, 1020–1025.

[46] Cuckoo Sandbox. [n.d.]. Hooked APIs and Categories in Cuckoo. ([n. d.]). https://github.com/cuckoosandbox/cuckoo/wiki/Hooked-APIs-and-Categories/.

[47] Igor Santos, Felix Brezo, Javier Nieves, Yoseba K Penya, Borja Sanz, Carlos Laorden, and Pablo G Bringas. 2010. Idea: Opcode-sequence-based malware detection. In *International Symposium on Engineering Secure Software and Systems*. Springer, 35–43.

[48] Marcos Sebastián, Richard Rivera, Platon Kotzias, and Juan Caballero. 2016. Avclass: A tool for massive malware labeling. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 230–253.

[49] SensorsTechForum. [n.d.]. Most Popular Windows File Types Used by Malware (2018). ([n. d.]). https://sensorstechforum.com/popular-windows-file-types-used-malware-2018/.

[50] FileHippo s.r.o. [n.d.]. File Hippo. ([n. d.]). https://filehippo.com/software/desktop.

[51] Ronghua Tian, Rafiqul Islam, Lynn Batten, and Steve Versteeg. 2010. Differentiating malware from cleanware using behavioural analysis. In *2010 5th international conference on malicious and unwanted software*. IEEE, 23–30.

[52] Dolly Uppal, Rakhi Sinha, Vishakha Mehra, and Vinesh Jain. 2014. Malware detection and classification based on extraction of API sequences. In *2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. IEEE, 2337–2342.

[53] Hashing Vectorizer. [n.d.]. Scikit-learn Machine Learning in Python. ([n. d.]). https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.HashingVectorizer.html.

[54] R Veeramani and Nitin Rai. 2012. Windows api based malware detection and framework analysis. In *International conference on networks and cyber security*, Vol. 25.

[55] Eunjin Kim Youngjoon Ki and Huy Kang Kim. [n.d.]. APIMDS-dataset. ([n. d.]). http://ocslab.hksecurity.net/apimds-dataset.

[56] Zhaoqi Zhang, Panpan Qi, and Wei Wang. 2019. Dynamic Malware Analysis with Feature Engineering and Feature Learning. *arXiv preprint arXiv:1907.07352* (2019).

A PSEUDOCODE OF DYNAMIC FEATURE EXTRACTION

Algorithm 1 Feature vector generation

Δ: Dataset of malware and benign behavior analysis reports [f_i]
processed_api_arg: List of the generalized API calls and arguments
Given: common_malware_types, common_registry_keywords and Δ
Results: (1) Feature vector of Method 1 [Feature_VectorM1], and Method 2 [Feature_VectorM2]
processed_api_arg = {}
foreach $f_i \in \Delta$ **do**
  Process the log file and extract its list of API calls (API_{ij}) and arguments (ARG_{ijk})
  Remove the suffix from the API name ['ExW', 'ExA', 'W', 'A', 'Ex'] in $API_{ij} \in f_i$
  foreach $ARG_{ijk} \in API_{ij}$ **do**
    switch (ARG_{ijk})
    Check if the common malware file types exists in command_line
    case $command\_line$:
      Call Algorithm 4
    Check if the regkey value is one of the common regkey for malware
    case 'regkey':
      Call Algorithm 3
    case 'path' **or** 'directory':
      Call Algorithm 5
    Remaining arguments with integer values, convert them into bin-based tags
    case IsNumber(ARG_{ijk}):
      Call Algorithm 2
    Remaining arguments with concrete values will not be changed
    else:
      processed_api_arg[ARG_{ijk}] = value(ARG_{ijk})
    end switch
  end foreach
  Features are constructed using Method 1 and Method 2 formulas
  M1processed_api_arg = Method1(processed_api_arg)
  M2processed_api_arg = Method2(processed_api_arg)
  Generate Method 1 and Method 2 feature vectors from the processed_api_arg using HashingVectorizer function
  Feature_VectorM1 = HashingVectorizer(M1processed_api_arg)
  Feature_VectorM2 = HashingVectorizer(M2processed_api_arg)
end foreach
return Feature_VectorM1, Feature_VectorM2

Algorithm 2 Logarithmic bin converter

Given: ARG_{ijk} is an argument with integer type
X = value(ARG_{ijk})
prefix = ' '
if value(X < 0 **then**
  prefix = 'neg_'
else if X = 0 **then**
  tag = 'num0'
else
  tag = 'numB'+ X
end if
return processed_api_arg[X] = prefix + tag

Algorithm 3 Registry key (common_key) extraction

Given: ARG_{ijk} is an argument
registry_tag = {}
foreach common_key in common_registry_keywords **do**
  if common_key = value(ARG_{ijk}) **then**
    registry_tag = common_key
  else
    registry_tag = other_reg
  end if
end foreach
return processed_api_arg[ARG_{ijk}] = registry_tag

Algorithm 4 Command line (CMD) extraction

CMD commonly appears in CreateProcessInternal
Given: ARG_{ijk} is an argument
c_l_tag = {}
foreach file_type in common_malware_types **do**
  if file_type = value(ARG_{ijk}) **then**
    c_l_tag = file_type
  else
    if not exists, return other_CL
    c_l_tag = other_CL
  end if
end foreach
return processed_api_arg[ARG_{ijk}] = c_l_tag

Algorithm 5 Directory and path extraction

Given: ARG_{ijk} is an argument
if value(ARG_{ijk}) = 'System32' **then**
  loc = 'sys_dir'
else
  loc = 'other_dir'
end if
return processed_api_arg[ARG_{ijk}] = loc

B DESCRIPTION OF USED MACHINE LEARNING ALGORITHMS

A description of each machine learning classifier is shown below:

- Support Vector Machine (SVM): uses a technique called the kernel trick to transform the data (N-dimensional space where N is the number of features). Based on these transformations, it finds an optimal boundary (hyperplance) between the possible outputs that distinctly classifies the data points [31].
- Decision Tree (DT): uses a tree-like diagram in which each internal node represents a test on a feature. It classifies the features by sorting them down the tree from the root to some leaf node where each leaf node represents a class label, and branches represent conjunctions of features that lead to those class labels. The paths from the root to the leaf represent classification rules [31].
- Random Forest (RF): consists of a large number of separate decision trees (DTs) that serve as an ensemble. Each

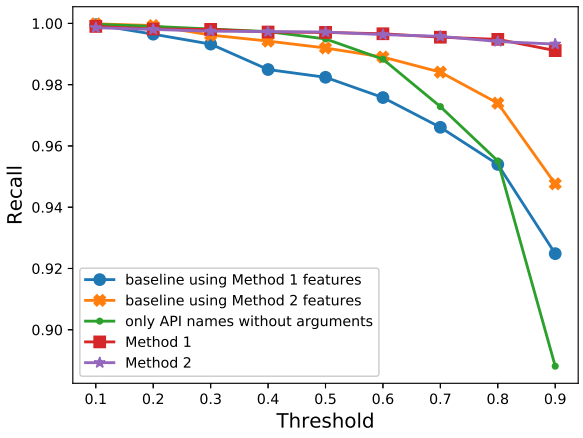

Figure 5: Comparison of our proposed methods with the baseline work [44], and [51] which uses only API names without arguments, in terms of recall scores over different threshold values.

tree in the random forest represents a class prediction. The class with the most votes in general becomes the concluding model's prediction [31].

- eXtreme Gradient Boosting (XGBoost): employs a decision-tree-based ensemble machine learning algorithm that uses a gradient boosting framework by combining a set of weak learners and making conclusions about the various feature importance and parameters [9]. Then it uses those conclusions to create new, more reliable models for regression and classification problems.
- Passive-Aggressive (PA): from its name, Passive means keep the model if the prediction is correct, and Aggressive means update the model to adjust to the misclassified instances if the prediction is incorrect. Consequently, it changes its classifier weight vector (update the classifier) for each misclassified case it receives, trying to correct it. Usually, it is used for classifying massive streams of data [12].

C MALWARE TYPES DESCRIPTION

A description of each malware type used in this paper is shown in Table 6.

D EVALUATION IN TERMS OF RECALL

Table 5 shows the performance comparison between our proposed methods against the six works from the state-of-the-art in terms of the F1 scores. This section presents the comparison based on another performance metric, recall, to evaluate the performance of our proposed methods against the state-of-the-art in terms of correctly classifying the malicious samples as malware. As shown in Table 5, [44] gives the highest F1 score among the six works. Thus, we use it as a baseline in our recall comparison experiments.

Table 6: A description of the malware types (forms) [28] used in this paper.

Malware Type		Description
Trojan		It hides as legitimate software and represents itself as something useful to deceive users into installing and downloading malware on their systems
Adware		It displays advertisements on the user's interface or screen automatically and usually within a web browser
Spyware		It secretly spies and gathers sensitive information without the user's permission and reports it to the spyware's author
Ransomware		It limits the access to the user's system by locking down the whole system or encrypting the files and then demanding to pay a ransom to get the files or system back and for its release
Virus		It replicates itself from device to device by modifying other device programs and poisoning them with its code
Worm		It spreads to other devices and usually causes harm by destroying files and data. However, unlike a virus, it has the ability of self-replicating and spreading itself independently without any need for human help
PUP		Potentially Unwanted Program: from the name, it is an unwanted program (e.g., advertising, toolbars, and pop-ups) that usually comes bundled with other irrelevant and legitimate software that has been downloaded by the user
Others	**Backdoor**	It bypasses the user's system security measurements and tries to gain a high-level privilege such as root access to install itself on a computer and thus allows the attacker to access it. Backdoor is generally classified as a Trojan
	Riskware	A program that is not designed to be strictly malicious but puts the user at risk in some way when it is used with bad intentions, as it has functions that can be used for malicious purposes (i.e., stealing data, causing disruptions or hijacking)
	Hack Tool	A special kind of Riskware program that is designed to hack into computers and networks

Moreover, to compare our work with other works that use different feature selection methods such as neglecting the API arguments and relying only on the API calls, we compare our methods with [51]. Figure 5 shows the comparison results of our proposed methods with the baseline work [44], and [51] which uses only API names without arguments, in terms of recall scores over different threshold values. As shown in the figure, for malware detection, our methods outperform the methods of the previous work [44, 51], where using our methods, fewer malware samples are misclassified as benign compared to their works.