electronics

MDPI

Article

A Streamlined Framework of Metamorphic Malware Classification via Sampling and Parallel Processing

Jian Lyu [1], Jingfeng Xue [1], Weijie Han [2], Qian Zhang [1,*] and Yufen Zhu [2]

[1] School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China; 3120195531@bit.edu.cn (J.L.); xuejf@bit.edu.cn (J.X.)
[2] School of Space Information, Space Engineering University, Beijing 101416, China; 3120185536@bit.edu.cn (W.H.); 3120205525@bit.edu.cn (Y.Z.)
* Correspondence: zhangqian16@bit.edu.cn

Abstract: Nowadays, malware remains a significant threat to the current cyberspace. More seriously, malware authors frequently use metamorphic techniques to create numerous variants, which throws malware researchers a heavy burden. Being able to classify these metamorphic malware samples into their corresponding families could accelerate the malware analysis task efficiently. Based on our comprehensive analysis, these variants are usually implemented by making changes to their assembly instruction sequences to a certain extent. Motivated by this finding, we present a streamlined and efficient framework of malware family classification named MalSEF, which leverages sampling and parallel processing to efficiently and effectively classify the vast number of metamorphic malware variants. At first, it attenuates the complexity of feature engineering by extracting a small portion of representative samples from the entire dataset and establishing a simple feature vector based on the Opcode sequences; then, it generates the feature matrix and conducts the classification task in parallel with collaboration utilizing multiple cores and a proactive recommendation scheme. At last, its practicality is strengthened to cope with the large volume of diversified malware variants based on common computing platforms. Our comprehensive experiments conducted on the Kaggle malware dataset demonstrate that MalSEF achieves a classification accuracy of up to 98.53% and reduces time overhead by 37.60% compared to the serial processing procedure.

Keywords: malware classification; malware family; parallel processing; microsoft kaggle malware dataset

check for updates

Citation: Lyu, J.; Xue, J.; Han, W.; Zhang, Q.; Zhu, Y. A Streamlined Framework of Metamorphic Malware Classification via Sampling and Parallel Processing. *Electronics* **2023**, *12*, 4427. https://doi.org/10.3390/electronics12214427

Academic Editor: Aryya Gangopadhyay

Received: 15 September 2023
Revised: 15 October 2023
Accepted: 24 October 2023
Published: 27 October 2023

Copyright: © 2023 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

In the digital age, networks became a prime target for many attackers. Malware is a prevailing weapon for attackers to launch network attacks and became a major challenge to cyberspace worldwide. In spite of the fact that anti-malware researchers put considerable efforts into the analysis task, it is still not ideal to curb malware attacks. Especially with high returns earned by malware, a consistent surge was found in malware attacks and obfuscated variants [1]. These variants are usually produced by modifying their binary codes or assembly instructions based on obfuscation techniques. Through the obfuscation process, the malware samples can change their structural characteristics and evade detection while preserving malicious functions [2]. The AV-TEST Institute reported that more than 350,000 fresh malware samples are discovered every day [3]. The proliferation of malware variants created a significant challenge for anti-malware analysis. To tackle the malware challenges, a significant amount of previous work was devoted to malware detection [4]. However, it is still a challenging issue to tackle with the obfuscated malware variants detection.

Motivated by the above-mentioned challenge, this paper focuses on the issues of malware variants classification aiming at accurately and effectively classifying the sheer number of variants into their families. Because it is a time-consuming workload to cope

with the large quantities of malware samples, the first issue we need to consider is to simplify the feature engineering phrase.

To extract features from malware, two traditional methods are often used: static analysis and dynamic analysis [5]. Behavioral information of malware is derived from a malware sample without actual execution in static analysis. On the contrary, a malware program needs to be run actually in dynamic analysis mode. Compared with dynamic analysis, static analysis can achieve high accuracy and efficiency due to being free from the overhead of execution cost [6]. Among the features extracted from static analysis, Opcode sequences garnered significant interest from malware researchers and are widely used in anti-malware analysis, which is because assembly instructions can reveal program behavior characteristics [7,8]. In order to accelerate the classification of a huge amount of malware, we decided to conduct static analysis and extract features from the assembly instructions. In addition to simplifying feature engineering, another strategy that we take into account is to employ parallel processing techniques to accelerate the classification of these massive malware variants by utilizing our ordinary computing platforms.

To this end, a light and parallel classification framework named MalSEF (a streamlined and efficient framework for detecting metamorphic malware families) is proposed in this paper. MalSEF mainly consists of four modules: sampling, feature extraction, feature matrix generation in parallel, and classification in parallel. MalSEF is implemented as follows. Firstly, select a small proportion of samples and construct a subset from the entire dataset according to a validated sampling criterion. Secondly, extract the Opcode sequence from the sampled subset to represent the entire dataset. Then, because the metamorphic technique is usually applied at the assembly instruction layer of malware, we analyze the behavioral characteristics based on the Opcode sequence, select the ranked Top-N Opcodes in frequency, and build a simple feature vector taking the frequency of Opcode as the eigenvalue to construct feature matrix in parallel for the initial dataset, based on the simple feature vector. The proposed parallel processing strategy is employed to reduce processing time and ameliorate processing efficiency for feature extraction and feature matrix generation from the entire dataset. By constructing the above lightweight feature set and applying the parallel processing strategy, the time overhead required for classifying large amounts of malware can be efficiently reduced. Finally, we evaluate MalSEF on the Microsoft Kaggle malware dataset and achieve promising classification accuracy. In addition, the processing time overhead can be reduced apparently compared with the serial processing mode. In conclusion, the following contributions are made in this paper:

(1) We suggest to extract a small proportion of samples from the entire dataset according to the selection criteria and construct a simple and efficient feature vector from the assembly (ASM) files that can reflect the original dataset. The final evaluations prove that the lightweight eigenvector can not only attenuate the complexity of feature engineering, but also satisfy classification requirements.
(2) We propose a parallel processing approach with commonly available hardware resources that utilizes collaboration of multi-core and active recommendation. The parallel strategy can run on the popular personal computer without high-performance hardware resources, and open the door for analysts to leverage general computers to tackle tough tasks due to the large volume of malware.
(3) We conduct systematic assessments using the Microsoft Kaggle malware dataset. The classification accuracy can reach up to 98.53%. The parallel processing technique results in a 37.60% reduction in the processing time compared with the conventional serial process mode. MalSEF can deliver a similar performance to the first winner of the challenge competition with the feature space effectively simplified, outperforming the existing algorithms in terms of simplicity and efficiency.

The remainder of the paper is organized as follows: Section 2 provides background information and summarizes related research; Section 3 describes the inspiration and framework of MalSEF; Section 4 presents the detailed implementation of MalSEF; Section 5

details the experiments and evaluations conducted on MalSEF; and the paper is concluded in Section 6.

2. Overview of Related Studies

A brief introduction of the necessary background knowledge and an overview of related research are provided in this section.

2.1. Background

2.1.1. Metamorphic Techniques

The purpose of metamorphism is to change its structure of the malware and generate new features at time of propagation, so as to avoid traditional feature-based detection [9]. Metamorphic techniques are quite widely used; some obfuscating method adopted in adversarial attacks [10] and backdoor attacks [11] can also be seen as metamorphism in a broader view, where metamorphism is usually implemented through resize and imperceptible perturbations [12]. As for malware, the metamorphic technique is usually applied in the assembly instruction layer, i.e., Opcode. The popular metamorphic methods include instruction reordering, dead/trash instruction insertion, and substitution [13]. The metamorphic technique is mainly applied at the assembly instruction level, which lies in the scope of this study (since the Kaggle dataset includes both assembly and binary formats of malware [14]). Below, we provide a brief overview of the common ways in which malware can be deformed.

(1) Instruction reordering

Instruction reordering involves division of code into segments, permutation of these segments, and inserting branch instructions as necessary to maintain the initial functionality. This code manipulation is extremely effective in modifying the initial signature and may also be effective against structure inspection techniques, such as detecting the change in entropy [15].

Instruction reordering can be realized in two ways. One way to accomplish instruction reordering is to randomize the sequence of instructions and remain the original control flow by inserting unconditional branches or jumps. As shown in Figure 1, the malware variant 'BKXtxeYlLsprabEWIQhn' in the Kaggle dataset employs this method to generate new variants. The other way is to swap independent instructions. Similar to compiling instructions, different compilers will generate different compilation instructions. The difference is that the goal of swapping instructions is to randomize the instruction stream.

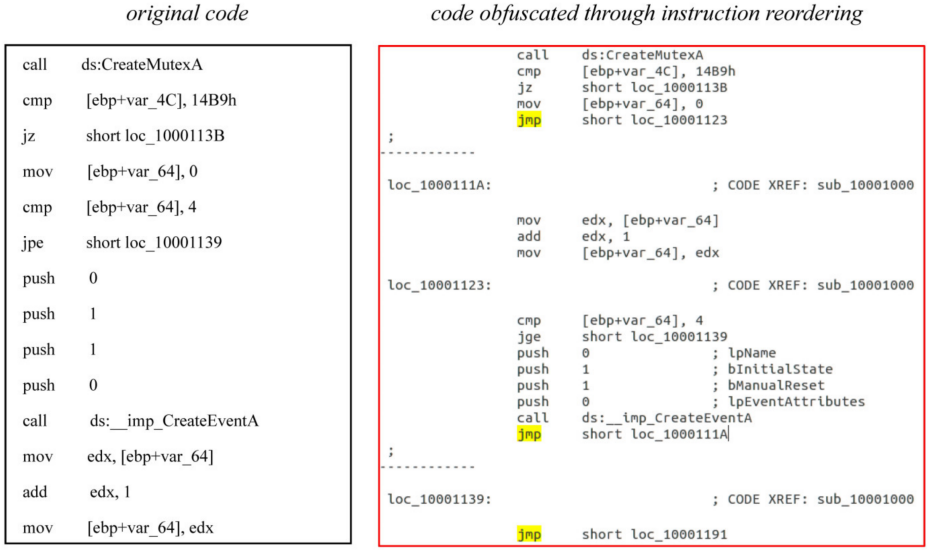


Figure 1. Example of metamorphism through instruction reordering.

This kind of metamorphism may interfere with the manual analysis process. However, many automatic analysis approaches that rely on intermediate representations, such as control flow diagrams or program dependency diagrams [16], can effectively overcome it because they are less sensitive to the unwanted changes in the control flow.

(2) Trash code or dead code insertion

Trash instructions are "not do anything" instructions that are executed without any impact on program functionality. Trash/dead instruction insertion adds code to a program without changing the program's original behavior. The simplest example is to insert a series of 'nop' instructions into a program. As shown in Figure 2, the malware variant 'i5u2KDJ9t0OyAdokafj7' in the Kaggle dataset employs this method to obfuscate and generate new variants.

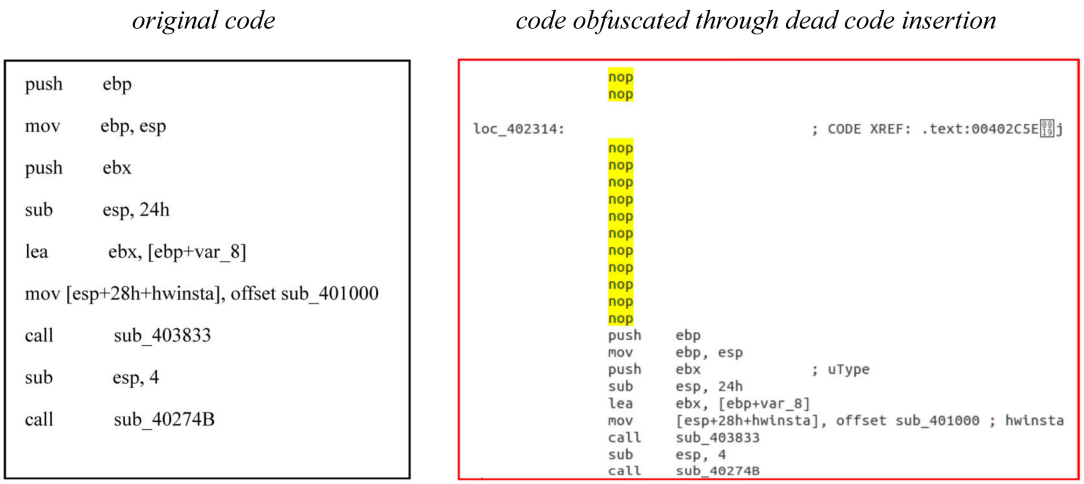


Figure 2. Example of metamorphism through dead code insertion.

Trash/dead instruction inserts are frequently used by malicious personnel to influence the detection based on features, and as for bypassing statistical detection, these metamorphic techniques are particularly effective.

(3) Instruction substitution

Instruction substitution means equivalent instruction replacement, which uses an equivalent instruction sequence dictionary to replace one instruction sequence with another.

The example of instruction substitution is illustrated in Figure 3. Because this modification relies on equivalent instruction knowledge, it makes the most serious obstacle to automatic analysis. The Intel Architecture 32-bit instruction set is abundant enough to perform the same operation in numerous ways. In addition, the IA-32 instruction set has some structural inconsistencies, such as a memory-based stack, which may be accessed by specialized instructions as a memory address. Operations of standard memory can also be accessed as a memory address, which makes the assembly language of IA-32 provide rich means for instruction replacement.

This kind of metamorphism is an effective way to evade feature-based detection and statistics-based detection. To tackle this form of obfuscation, an equivalent instruction sequence dictionary, which needs to be similar to the one used in generating the equivalent instruction, must be maintained for the analysis tool. This is not a fundamental way to tackle the instruction replacement problem, nevertheless, it is usually enough to handle usual situations.

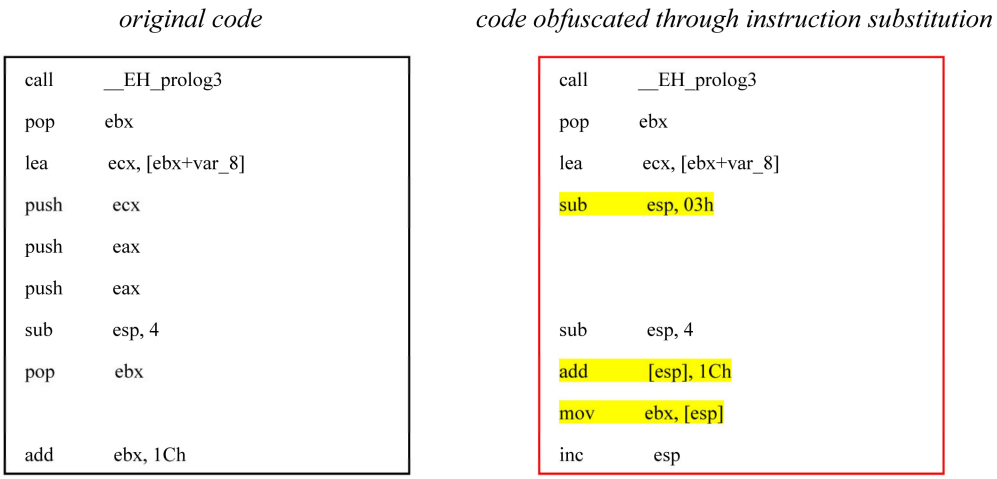


Figure 3. Example of metamorphism through instruction substitution.

2.1.2. Methods for Malware Analysis

In the field of malware analysis, two fundamental tasks are malware detection and malware classification [17]. To find malicious samples from unknown programs is the target of malware detection, while to separate malware into corresponding families is the target of malware classification. Features of programs extracted in the detection stage can still be used in the classification process. Based on the different ways of program feature extracting, methods of malware detection and classification are put into two categories: static and dynamic.

Static analysis does not run the program actually. To understand the malicious nature of a program, researchers usually gather information from its PE header, PE body, and binary code. Alternatively, they can disassemble the binary code and extract Opcodes or other pertinent details from the assembly program to characterize it [18]. The static analysis technique is more efficient but must contend with packaging and obfuscation disturbances.

Dynamic analysis is an approach based on behavior that needs to run the program to capture its run time behavior characteristics. The main behavior characteristics of the dynamic analysis method include program API sequence and behavior interaction with OS during run time. In order to avoid damage to the terminal system caused by malware, a virtual environment is usually used while dynamic analysis is performed [19,20].

Additionally, combining static and dynamic analysis, there are also researchers that conducted hybrid analysis. During the hybrid analysis, static and dynamic features are derived separately, and then fused together to overcome the defects of static or dynamic analysis alone, so as to achieve a more comprehensive and accurate analysis of malware [21,22].

2.1.3. Parallel Processing Techniques

Parallel processing is a technique that allows for the execution of multiple tasks simultaneously by dividing a serial work process into multiple processes or threads [23–25]. Some common parallel processing methods are as follows:

(1) Task parallel: during a complete working process, if there are some independent modules executing in parallel, this parallel processing method is called task parallelism. As shown in Figure 4, in this data flow diagram, when module A and B execute in parallel, it is called task parallel.
(2) Pipeline parallel: when a series of connected modules (forming a complete working process) process independent data elements in parallel (these data elements are usually a time series or an independent subset of a certain dataset), this parallel processing method is called pipeline parallelism. As shown in Figure 4, when modules A, C, and D execute in parallel, it is called pipeline parallel.
(3) Data parallel: when a dataset can be divided into a number of subsets and these subsets can be processed simultaneously, this kind of parallel processing method is

called data parallelism. As shown in Figure 4, if module B reads data in parallel, it is called data parallel.

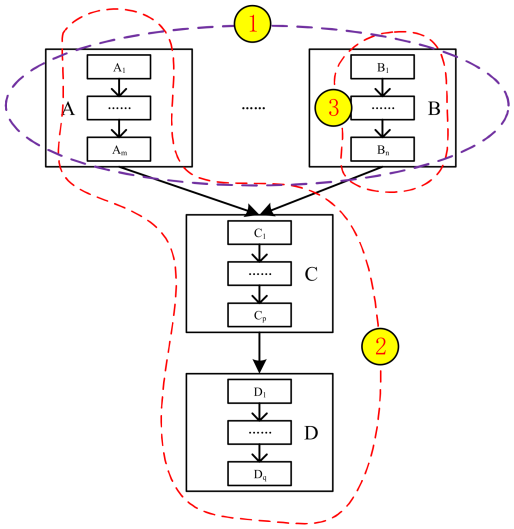


Figure 4. A data flow diagram.

2.2. Related Studies

Since the target of this research is malware family classification based on parallel processing technology, this section makes a brief summary of the relevant research work on malware family classification and malware analysis using parallel technologies.

In the process of malware classification, researchers usually extract different features to represent samples, and then select an automatic classifier to realize classification. For example, Bailey Michael et al. [26] proposed to describe malware behavior characteristics according to system state changes (such as file reading and writing, process creation, etc.). On this basis, malware is automatically classified into groups with similar behavior types in automatic mode, so as to handle the phenomenon of the sharp increase in malware scale and behavior difference. In addition, some researchers also proposed methods to analyze the binary code of malicious programs. Nataraj Lakshmanan et al. [27] applied the binary texture analysis method to analyze the malicious code and made a comprehensive comparison. The authors found that the binary texture analysis method not only achieves relatively high classification accuracy, but also surpasses the dynamic analysis by nearly 4000 times in speed. Furthermore, some of the current strategies can even be resisted by texture-based analysis.

In view of the sharp increase in the scale of malware variants, Ahmadi Mansour et al. [28] designed a novel classification strategy for malware family, utilizing the dataset of the Microsoft malware challenge as the research object. An innovative approach for feature set extraction and selection was devised to effectively characterize malware samples. According to the different characteristics of malicious code, the features were classified and weighted. This method can effectively address the challenge brought by the increasing variants of malware. To cope with the challenge of the increasing number of malware from an extensible perspective, Hu Xin et al. [29] proposed a machine learning framework that utilizes diverse content features (such as string, instruction sequence, section information, and other characteristics) and gathers intelligence information from external resources (such as anti-virus output). Several optimization methods were designed to further improve the performance of the classifier. Finally, an experiment was conducted on the Kaggle dataset of the Microsoft malware challenge to evaluate the performance of the propounded approach in processing malware datasets with a large scale. Considering that most malicious programs are variants created from existing malware, Lee Taejin et al. [30] presented a method for malware detection and classification in light of the local clustering

coefficient. In this way, the classification reliability can be enhanced, the automatic selection and management of the malware family can be realized, and the large-scale malware can be classified efficiently.

Aiming at overcoming the limitations of current malware classification that requires domain knowledge, Raff Edward et al. [31] used a new representation named the SHWeL feature vector through expanding the concept of Lempel-Ziv Jaccard distance. The SHWeL vector improves the accuracy of LZJD, and its performance is better than the byte n-grams feature representation method, which can build an effective eigenvector input for classifier training and reasoning. In addition, Quan Le et al. [32] also presented a method of malware classification that does not require domain knowledge. This method converts a binary file into a one-dimensional representation and then constructs a deep learning network for automatic classification. The advantage of this method is that researchers do not need to extract features from malware, but only use the universal image scaling algorithm to convert a malware into a one-dimensional form of fixed size.

With the increase in the scale of malware, the analysis workload becomes more and more onerous. To ameliorate the efficiency of malware analysis, some researchers began to apply parallel and distributed processing technologies.

Junji NAKAZATO et al. [33] designed a new malware classification method to tackle the problems that existing methods are inadequate for achieving efficient and accurate classification. The authors initially conducted a dynamic analysis to automatically extract execution traces of malware, then employed their behavioral characteristics to classify them into distinct categories. In the process of obtaining behavior characteristics, the author adopted a parallel processing method to extract the API information of malware. Sheen Shina et al. [34] derived features of different categories from the executable file and applied them to the integrated classifier. The integration approach combined several separate pattern classifiers for better results. The problem was how to select classifiers of a minimum number and acquire the best results. This paper designed a compact integration technique using the harmony search method, which was derived from the music harmony-inspired algorithm. For malware detection, the simplified integrated classifier was used. For selecting the optimal subset of classifiers, multiple variegated classifiers were integrated into harmony and parallel search. Wang Xin et al. [35] used parallel machine learning and information fusion technology to achieve more effective detection. The author first extracted eight types of static features from the program and constructed two sets of features through feature selection; subsequently, a parallel machine learning model was employed to expedite the classification process. Finally, information fusion was achieved through probability analysis and Dempster–Shafer theory to complete the final detection.

Based on the above-introduced literature, the researchers made multi-aspect attempts at coping with the sharp increase in the scale of malware variants; nonetheless, there still exist apparent deficiencies as follows (as illustrated in Table 1):

(1) The feature vectors constructed by these methods can usually effectively represent the features of malware. However, when applying to analyze massive malware, the feature space will become too large and bring a heavy burden to analysts. For example, the feature vector proposed by [28] can achieve perfect classification accuracy; however, its feature vector is too large to be realized under the condition of ordinary computing resources.

(2) The parallel processing method above is usually applied to multiple classifiers for parallel detection. In essence, it is still a serial processing mode, and the parallel analysis target of massive samples is not achieved. When faced with a large scale of malware, it will inevitably affect the analysis efficiency.

Table 1. Pros and cons of similar approaches for malware family classification and parallel processing.

Related Work	Analysis Method	Dataset	Feature Set	Pros	Cons
Bailey et al. [26]	Dynamic analysis	Network security community and a part of the Arbor Malware Library	system state changes	Offer an innovative perspective on comprehending the connections between malware	The underlying shortcomings due to dynamic analysis
Nataraj Lakshmanan et al. [27]	Static analysis	Host-Rx dataset, Malhuer dataset and a VX Heavens dataset	binary texture feature	With comparable classification accuracy, faster than dynamic technique	Lack of the semantic analysis of the binary programs
Ahmadi Mansour et al. [28]	Static analysis	Microsoft Malware Challenge dataset	features extracted from hex dumps and decompiled files	High classification accuracy	Complicated feature matrix and high time consumption
Hu Xin et al. [29]	Static analysis	Microsoft Malware Challenge dataset	multifaceted content features and threat intelligence	High classification accuracy	The process of feature extraction will require a large amount of time
Lee Taejin et al. [30]	Dynamic analysis	Malware collected from a commercial environment	System call sequences	Good dependability on processing malware volume	May fail to yield an accurate result; high time consumption due to dynamic analysis
Raff Edward et al. [31]	Static analysis	Microsoft Kaggle dataset and Drebin dataset	SHWeL feature vector by extending Lempel-Ziv Jaccard Distance	Not require domain knowledge	High time consumption
Quan Le et al. [32]	Static analysis	Microsoft Kaggle dataset	One dimensional representation of the malware sample	Not require domain knowledge	Lack of the semantic analysis of the binary code due to the black-box feature of the deep learning model
Junji Nakazato et al. [33]	Dynamic analysis	Not clearly stated	Dynamic execution traces	Extract API calls in parallel thread	The parallel processing is not clearly stated
Sheen Shina et al. [34]	Hybrid analysis	Not clearly stated	PE features and API calls	Construct the at least as good as ensemble classifiers in parallel fashion	Not apply parallel technique into the analysis process
Wang Xin et al. [35]	Static analysis	Drebin and Android Malware Genome Project	Eight types of static features	Achieve higher detection accuracy	Not apply parallel technique into the analysis process

3. Overview of MalSEF

3.1. Motivation

The malware detection process typically consists of two main stages: feature extraction and detection/classification, which constitute a serial process [6]. Malware detection and malware classification are two distinct tasks. The former aims to determine whether an unknown sample is malicious, while the latter involves grouping detected malware into its most appropriate family. In the current cyberspace environment, we are witnessing a surge in malware families, coupled with a high number of variants within each family, posing a significant challenge to the anti-malware community [2].

Actually, the scale of malware volume is increasing mainly due to the widespread application of metamorphic technologies. The metamorphic malware will modify its code structurally while maintaining its functionality at time of propagation. How to cope with the large volume of metamorphic malware variants and group them into appropriate families became a crucial task for the security community. During the grouping process, the following challenges need to be addressed:

(1) The complexity of feature engineering may increase sharply due to the large scale of malware variants. The complexity is mainly derived from two aspects: (1) The feature set of the cutting-edge analysis methods is usually fairly complicated, because we often utilize APIs or Opcodes, or their combination (n-grams) as the feature vector to profile the malware. The number of API and Opcode on the Win32 platform is relatively large. (2) The increasing volume of malware variants will further aggravate the complexity, especially in the current massive malware environment.
(2) The efficiency of coping with the sheer number of malware samples cannot be guaranteed. The existing detection methods mainly include the training stage and detection stage. These two stages are separately implemented and the detection process is essentially a serial processing one. We may inevitably fail to tackle the challenge facing the environment of large-scale malware.

Driven by the above issues, we put forward the following ideas in this paper:

(1) How to establish a simple feature set for the large number of malware variants so as to alleviate the computation cost and deliver satisfactory classification performance simultaneously?
(2) How to efficiently handle numerous malware variants and classify them into their homologous families to ensure the efficiency of the classification process?
(3) How to implement the complicated task for analyzing a huge amount of malware on a simple personal computer, so as to enable ordinary researchers to accomplish the seemingly impossible task due to the conventional paradigm?

The answers to these questions were already provided in previous studies. Motivated by the above inspiration, we aim to propose a lightweight approach for ordinary researchers to perform heavy analysis tasks using common computing platforms.

3.2. Parallel Processing Model of Massive Malware Classification

Based on the principle of parallel processing technology, the procedures in malware detection that can be optimized by parallel processing include:

(1) In the training stage, the process of extracting features from the training set include "assembly commands of samples → extracting Opcode lists → counting the occurrences of every Opcode → generating feature vectors → generating feature matrices", which can be implemented in parallel mode.
(2) In the detection stage, the process performing on unknown samples include "assembly commands of samples → generating feature vectors → classification", which can be implemented in parallel mode.

By considering the above malware analysis process, we employ parallel processing techniques to compress the needed analysis time and boost the malware classification

efficiency. According to the above discussion, the malware classification process in parallel is designed as displayed in Figure 5.

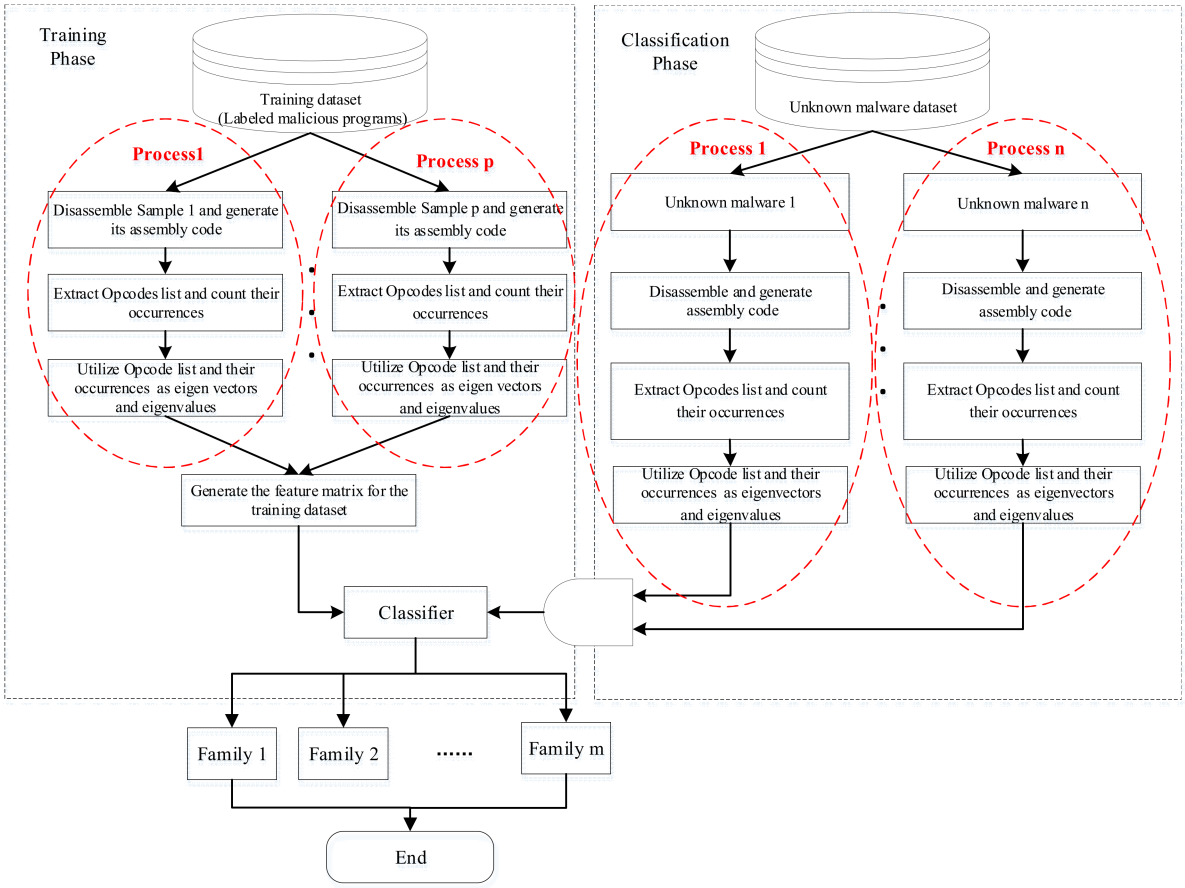


Figure 5. Parallel classification model of massive malware.

3.3. Overall Framework of MalSEF

MalSEF mainly includes four modules, namely, sampling of massive samples, feature extraction, feature vector generation, and classification, as shown in Figure 6.

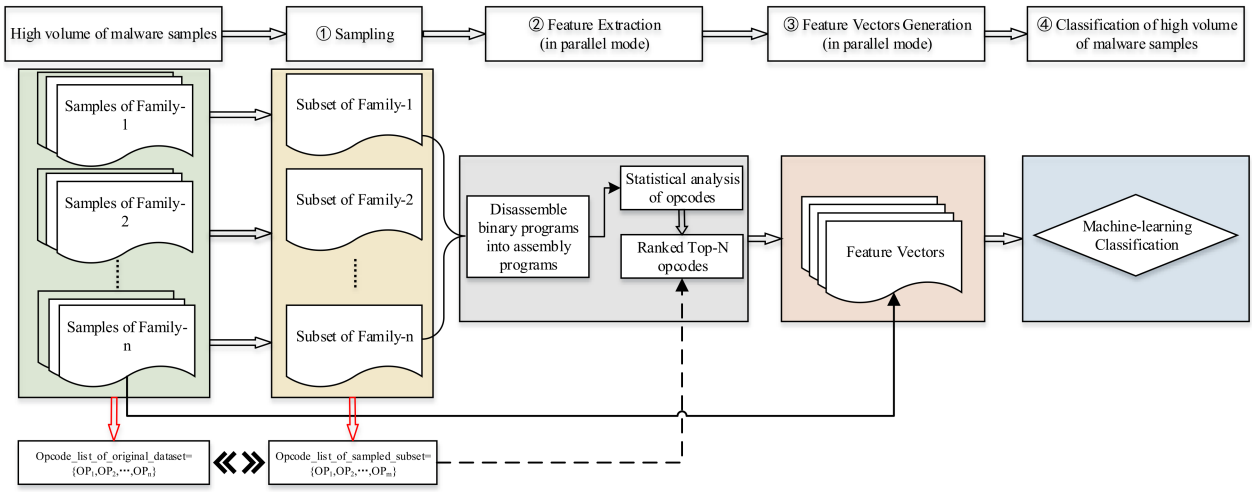


Figure 6. The framework of MalSEF.

(1) Sampling of massive samples

The task of this module is to construct sampled subsets that consist of a small part of samples randomly selected from each family of the massive samples collected, which will be used as the data source for feature extraction. The purpose of this module is to select features from subsets of samples to represent the original dataset rather than extract features directly from massive samples, so as to attenuate the complexity of the extraction of features. The subset size is determined on the criteria of selecting a smaller quantity of samples from the entire dataset and obtaining the same results that reflect the original dataset as previously as needed [36].

(2) Feature extraction

Because the malware program is often in the form of binary code, we have to firstly convert the binary malware programs into assembly programs. The most frequently used disassembling tool is IDA Pro. In our study, because the experiment dataset of this paper is released by Microsoft in Kaggle, which is in the form of the assembly instructions after disassembling, accordingly, features can be extracted statically from the assembly programs of malware samples. This module first counts the occurrences of every Opcode in the sample subsets and then ranks Opcodes according to their occurrence frequencies. Finally, the Top-N ranked Opcodes are selected as the feature vector (Top-N is set by researchers flexibly) and the number of occurrences of each Opcode is taken as its feature value.

(3) Feature matrix generation

After generating feature vectors in the previous module, the task of this module is to generate the feature matrix for the samples. According to the experimental method of cross-validation, we need to generate feature matrices for the samples in the training set and testing set, respectively. Among them, the training set is a labeled dataset, and the test set is an unlabeled dataset. Since the number of samples in the testing set is much larger than that in the training set, it requires more time to construct feature matrices for the testing set. To this end, we adopted a parallel processing approach to generate feature matrices for the testing set.

(4) Classification of massive malware samples

Having gotten the feature matrix as illustrated above, we can verify the result of MalSEF by cross-validation.

4. Detailed Implementations of MalSEF

This section describes the implementation of MalSEF in detail.

4.1. Sampling from the Massive Samples Set and Feature Extraction

4.1.1. Sampling from the Massive Sample Set

The aim of this module is to select a certain number of samples from each family to form a subset for each family. The size of each subset is determined by a sample size determination criterion. The required sample size is decided based on the following formula.

$$n_0 = \frac{z^2 \times p \times q}{d^2} \tag{1}$$

where: n_0 is expected sample size; z is standard normal deviation of expected confidence level; p is the assumed proportion of the target population with an estimated particular characteristic; $q = 1 - p$; and d is the degree of accuracy expected in estimated proportion. This is a theoretical formula that is used to determine the number of selected samples n_0 under a certain expected confidence level and certain marginal error, allowing n_0 to represent the overall samples in a statistical sense. It is based on the assumption that the samples follow a normal distribution, noting that the expected confidence level is determined by z (e.g., z = 1.96 for 95% confidence level), and the marginal error is determined by d (e.g., d = 0.01 for a margin of error of 1%). As for p, when z and d are given

and the estimator p is not known, $p = 0.5$ can be used to yield the largest sample size, which is the p value we used in this research.

The formal description of the sampling process is as follows:

$$Set(S\begin{bmatrix} F_1(f_{11}, f_{12}, \cdots, f_{1N_1}) \\ \vdots \\ F_m(f_{11}, f_{12}, \cdots, f_{1N_m}) \end{bmatrix}) : Sampling :\rightarrow Subset(S'\begin{bmatrix} F_1\left(f_{11}, f_{12}, \cdots, f_{1N_1}{}'\right) \\ \vdots \\ F_m\left(f_{11}, f_{12}, \cdots, f_{1N_m}{}'\right) \end{bmatrix}) \quad (2)$$

where ${N_1}' \leq N_1, \cdots, {N_m}' \leq N_m$, and $S' \subseteq S$.

Description of the algorithm implementation of the sampling process is illustrated as Algorithm 1.

Algorithm 1: Sample a subset from the original dataset

//S represents the original dataset.

//$\overbrace{z}$ represents the set of standard normal deviations for desired confidence level of original dataset.

//$\overbrace{p}$ represents the set of assumed proportions in the target population of original dataset.

//$\overbrace{\mathrm{d}}$ represents the set of degrees of accuracy desired in the estimated proportion of the original dataset.

Input: the original malware dataset S, $\overbrace{z}$, $\overbrace{p}$, $\overbrace{\mathrm{d}}$
Output: the sampled malware subset S′

```
trainLabels = readDataset() //read the original dataset
labels = getLabels(S) //read the labels of the original dataset
for i = 1 to labels do
    mids = trainLabels[trainLabels.Class == i] //get the samples of Class == i
    mids = mids.reset_index() //reset the index of the samples of Class == i
    n_i = (z_i^2 × p_i × (1 − p_i)) / d_i^2 //calculate size of the subset for the ith original dataset
    for j=1 to n_i do
        rchoice = randit(0, n_i) //select digits from 1 to 100 randomly
        rids = mids [1, rchoice] //build the subset for the Class = i
    end for
    S'.append(rids) //append the subset of Class == i to S'
end for
return S'
```

4.1.2. Feature Extraction and Feature Vectors Construction

Opcode is a mnemonic of machine code, which is usually represented by assembly command [2]. Because this paper is based on a dataset released by Microsoft in Kaggle, which contains the assembly program of malware, we build the features of samples by extracting Opcodes from the assembly program. The procedure of extracting features using the sampled subset is as follows:

(1) Analyze samples in the subset one by one and extract Opcodes from each sample;
(2) Count the occurrences of each Opcode in the subset;
(3) Sort Opcodes based on their occurrences;
(4) Select Top-N Opcodes as the feature vector (the value of Top-N is set by the researchers flexibly).
(5) Construct the feature matrix by counting the occurrence numbers of each opcode in each set as the feature value for the feature element.

The description of the implementation process of feature extraction and feature vectors construction is illustrated as Algorithm 2.

Algorithm 2: Feature matrix construction

Input: assembly programs of the malware samples $P = \{p_1, p_2, p_3, \cdots, p_n\}$
Output: the feature matrix FM

```
for i=1 to Len(P) do
    //Append the opcodes of the analyzed program to the Opcode list
    OpList ← Opcodes of p_i
    //Build the opcode sequence of all the programs
    OpSeq[P⃗] ← OpListofp_i
    //Count the occurrence times of each Opcode
    OccSeq[P⃗] ← OpSeq[P⃗]
end for
//Sort the Opcode sequence based on occurrences
Sorted_OpSeq[P⃗] ← OpSeq[P⃗](Key : OccSeq[P⃗])
//Select the ranked Top-N Opcodes as the feature vector and their occurrences as the feature values
FM ← Ranked[Sorted_OpSeq[P⃗]]
return FM
```

4.2. Feature Extraction with Multi-Core Collaboration and Active Recommendation in Parallel

Parallel processing in the procedures of massive malware detection has the following characteristics:

(1) Within the two procedures of parallel processing described in Section 3.2, the relationship between tasks in each procedure is loosely coupled.
(2) In the two procedures of parallel processing described in Section 3.2, we should adopt the iteration method to tackle the distributed parallel tasks load in the practical processing.
(3) In the procedure of parallel processing, different tasks will inevitably lead to different running speeds because of the different performance of the running environment and different computational load. Thus, tasks should be adaptively distributed according to the practical performance of each running node.

Based on the characteristics of the above procedures of parallel processing, we consider using the idea of multi-core to build multiple task nodes to extract code features in parallel. According to the practical implementation of each task, we can actively assign processing tasks to better performance nodes or faster nodes, so as to achieve parallel feature extraction of massive malware.

To this end, we design a parallel processing method with collaboration among multi-cores and active recommendation. Figure 7 illustrates the procedure, and the comprehensive implementation process is as listed below:

(1) Construct the analysis sample queue based on the malware sample sets. The queue is illustrated as follows:

$$Queue_{sample} = \{sample_1, sample_2, \cdots, sample_n\}. \quad (3)$$

(2) Construct a multi-core resource pool based on the computing resource nodes, each of which serves as a processing core. The core pool is illustrated as follows:

$$Queue_{core} = \{core_1, core_2, \cdots, core_m\}. \quad (4)$$

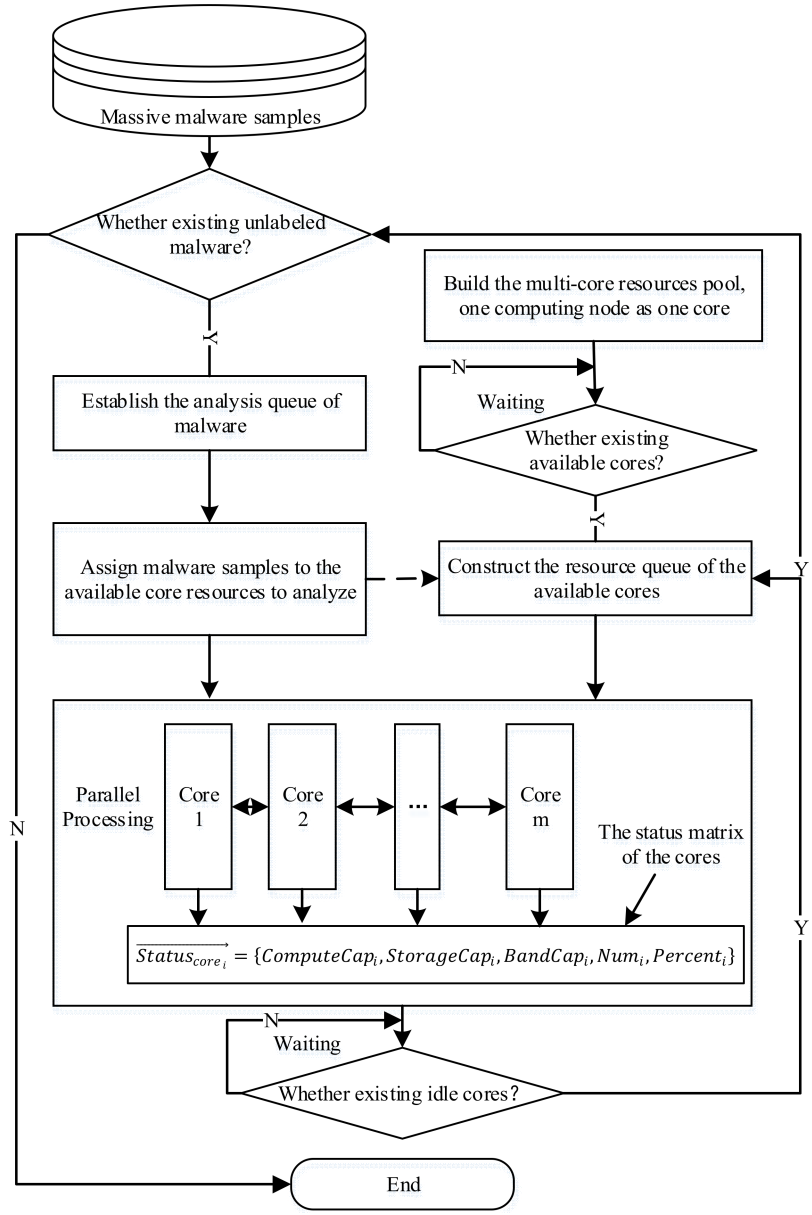


Figure 7. Feature extraction with multi-core collaboration and active recommendation in parallel.

(3) Query the current available resources in the resource pool and establish a queue of current available resources. The available resource queue is depicted as below:

$$Queue_{core'} = \{core_1, core_2, \cdots, core_{m'}\} (1 \le m' \le m). \quad (5)$$

(4) According to the currently available resource queue, the master node (the node that stores the sample sets) fetches m' samples from the sample sets and allocates them to the resource queue for processing.

(5) In the procedure of task parallel processing, each node monitors its own running state in real time, maintains and updates its own state vector in real time, and communicates the state vector information with each other. Based on the real-time interactive state vector information among the cores, the state vectors are combined to form a global state matrix within the cores.

The state vector describes the progress of current task processing, computational resource consumption, storage resource consumption, bandwidth resource consumption,

and the number of tasks that were assigned in a node. The state vector is illustrated as follows:

$$\vec{Status}_{core_i} = \{ComputeCap_i, StorageCap_i, BandCap_i, Num_i, Percent_i\}. \quad (6)$$

The elements in the state vector are defined as follows:

(1) $ComputeCap_i$: computational resource consumption of node i;
(2) $StorageCap_i$: storage resource consumption of node i;
(3) $BandCap_i$: bandwidth resource consumption of node i;
(4) Num_i: the number of tasks assigned to node i;
(5) $Percent_i$: the progress of current task processing of node i.

In order to ensure a reasonable allocation of the subsequent tasks when combining the state vectors of each node to construct the state matrix, it is essential to rank the nodes based on their real-time status. Nodes with a lighter load are positioned higher, while those with a heavier load are placed lower.

To this end, after receiving the state vectors from other nodes, each node compares the values of each element in the state vector according to the importance order of "$Percent > ComputeCap > Num > StorageCap > BandCap$", and forms the final state matrix, which is described as follows:

$$[Status] = \begin{bmatrix} \vec{Status}_{core_1} \\ \cdots \\ \vec{Status}_{core_{m'}} \end{bmatrix} = \begin{bmatrix} ComputeCap_1 & StorageCap_1 & BandCap_1 & Num_1 & Percent_1 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ ComputeCap_{m'} & StorageCap_{m'} & BandCap_{m'} & Num_{m'} & Percent_{m'} \end{bmatrix}. \quad (7)$$

(6) The master node will continuously monitor the state matrix and sort the samples to be analyzed. Once a node finishes its task, it notifies the master node, which then assigns new samples to that node, allowing it to start a new processing task. By this way, our method realizes the real-time and active pushing of processing tasks.
(7) Run continuously until the samples are processed entirely according to the above scheme.

5. Evaluation

5.1. Experimental Configuration

Hardware: Lenovo ThinkStation, Intel® CoRE™ i7-6700U CPU@3.40GHz×8, 8 GB memory (Made by Lenovo in China).

Operating system: 64 bit Ubuntu 14.04.

5.1.1. Dataset

In the evaluation phase, we use the training dataset released by Microsoft in Kaggle, which includes 10,868 malicious samples. We carry out cross-validation on the training dataset directly because we could not obtain the labels of the testing samples. Each malware sample has a 20-character hash value as its ID. The dataset comprises nine distinct malware families, namely Ramnit (R), Lollipop (L), Kelihos ver3 (K3), Vundo (V), Simda (S), Tracur (T), Kelihos_ver1 (K1), Obfuscator.ACY (O), and Gatak (G). Each sample family is assigned a class label from 1 to 9. The training dataset is shown in Table 2, which illustrates the distribution of each malware family.

5.1.2. Classifier

During the malware classification stage, we use data mining techniques to automate the task. To evaluate the effectiveness of MalSEF, we apply four different classifiers for identifying malware, including decision tree (DT), random forest (RF), K-nearest neighbor (KNN), and extreme gradient boosting (XGB), which are commonly used in the fields of data mining, information security, and intrusion detection [37].

Table 2. Experimental dataset.

Family ID	Family Name	#
1	Ramnit (R)	1541
2	Lollipop (L)	2478
3	Kelihos ver3 (K3)	2942
4	Vundo (V)	475
5	Simda (S)	42
6	Tracur (T)	751
7	Kelihos_ver1 (K1)	398
8	Obfuscator.ACY (O)	1228
9	Gatak (G)	1013
Total		10,868

5.2. Experimental Results and Discussion

We evaluate MalSEF comprehensively through detailed experiments in this section. Firstly, we build Opcode lists from the original samples set and the sampled subset separately and evaluate the classification effect of the total sample set using different Top-N Opcode lists, which are selected to construct feature vectors. Secondly, we apply the parallel processing technology in the procedure of classification based on the origin sample set and sampled subset separately, and verify the effect of parallel classification. Finally, we compare MalSEF with similar studies comprehensively.

5.2.1. Classification Using Features Derived from the Original Set and the Sampled Subset

This section indirectly evaluates the effect of using a sampled subset to characterize the original dataset. Features are extracted from the original set and the sampled subset, respectively, and then we classify the Train dataset to evaluate its classification effect.

Composition of the Sampled Dataset—Subtrain

In the experimental stage, we first select a small part of samples from the whole experimental dataset to form the subset, which is used to construct feature vectors. Because the experimental dataset we applied in this research is the Train dataset from Microsoft in Kaggle, we named this subset as Subtrain. Subtrain consists of 803 samples with specific information as shown in Table 3.

Table 3. Subtrain: the sampled dataset from the Train dataset.

Family ID	Family Name	#
1	Ramnit (R)	95
2	Lollipop (L)	100
3	Kelihos ver3 (K3)	97
4	Vundo (V)	96
5	Simda (S)	39
6	Tracur (T)	92
7	Kelihos_ver1 (K1)	91
8	Obfuscator.ACY (O)	97
9	Gatak (G)	96
Total		803

Evaluation of the Classification Results Using Features Extracted from the Entire Train Dataset

By analyzing the dataset released by Microsoft in Kaggle, we find that there are 735 Opcodes in the Train dataset. To verify the impact of the 735 Opcodes on classification results, we rank them according to their occurrences and select Top-N Opcodes as the feature vector to verify the final classification effect.

We first select various Top-N Opcodes to generate feature vectors, followed by conducting 10-fold cross-validation experiments using random forest (RF), decision trees (DT), support vector machine (SVM), and extreme gradient boosting (XGBST) classifiers. We utilize the open-source Python library Scikit learn (version 1.0.2) and opt for the default values when it comes to setting the hyper parameters for various classification methods which makes it easier to ensure the generalization ability of MalSEF without intentional optimization. The classification outcomes are presented in Table 4.

Table 4. Evaluation of the classification results according to various Top-N Opcodes extracted from the Train dataset.

	RF	DT	SVM	XGBST	Number of Feature Opcodes
Accuracy	98.34%	97.06%	97.38%	98.16%	N = 735
Precision	97.93%	92.13%	97.17%	96.29%	
Recall	97.83%	94.92%	93.63%	97.57%	
F1 Score	97.77%	93.17%	95.02%	96.80%	
Accuracy	98.20%	97.52%	97.65%	98.21%	N = 400
Precision	97.62%	94.20%	90.58%	93.90%	
Recall	91.00%	90.31%	90.31%	89.69%	
F1 Score	92.67%	91.49%	90.35%	90.69%	
Accuracy	98.57%	97.52%	97.19%	98.39%	N = 300
Precision	98.26%	91.11%	89.18%	97.98%	
Recall	93.65%	90.20%	92.39%	95.69%	
F1 Score	95.34%	90.56%	90.29%	96.65%	
Accuracy	98.44%	97.15%	96.55%	98.30%	N = 200
Precision	97.97%	96.48%	87.77%	97.81%	
Recall	93.30%	90.82%	88.87%	93.16%	
F1 Score	94.85%	92.36%	88.08%	94.72%	

As can be seen from Table 4, results of classification are similar using different Top-N Opcodes. The classification effect of the RF classifier with different Top-N opcodes is as shown in Figure 8, where Train_N stands for Top-N Opcodes extracted from the Train dataset and the fold line at the top portrays the variation in accuracy when the selected number of opcodes is changed.

The results of this experiment demonstrate that selecting all Opcodes as feature vectors is not necessary to represent the characteristics of samples. Because there exists redundancy in the extracted Opcode lists, only a part of key Opcodes is needed to achieve the desired classification effect.

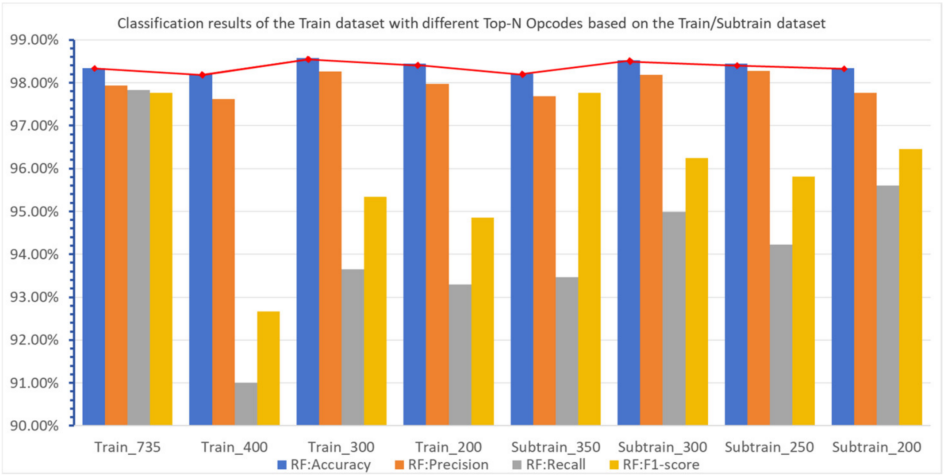


Figure 8. Comparison of the classification results of RF classifier with different Top-N Opcodes based on Train/Subtrain dataset.

Evaluation of the Classification Results Using Features Extracted from the Sampled Subtrain Dataset

In order to verify whether the sampled set Subtrain can comprehensively and effectively represent the features of the Train set, in this section, we use the sampled Subtrain set as a data source to count the occurrences of all Opcodes and sort them. Subsequently, various Top-N opcodes are chosen as the feature vector to create feature matrices for the original Train set. Finally, we carry out cross-validation experiments on the Train set to verify the validity of using the sampled subset to classify the original set.

There are 394 different Opcodes extracted from the Subtrain set, and we chose different Top-N Opcodes to classify the Train set separately. Outcomes of the experiment are presented in Table 5.

Table 5. Evaluation of the classification results according to various Top-N Opcodes extracted from the Subtrain dataset.

	RF	DT	SVM	XGBST	Number of Feature Opcodes
Accuracy	98.21%	97.24%	96.87%	98.39%	N = 350
Precision	97.69%	96.75%	91.61%	96.20%	
Recall	93.47%	89.73%	92.11%	93.67%	
F1 Score	95.01%	91.54%	91.71%	94.67%	
Accuracy	98.53%	97.38%	96.83%	97.98%	N = 300
Precision	98.18%	93.76%	92.16%	97.74%	
Recall	94.99%	90.53%	90.32%	91.43%	
F1 Score	96.25%	91.57%	90.98%	93.24%	
Accuracy	98.44%	97.29%	96.83%	98.34%	N = 250
Precision	98.27%	94.04%	93.44%	98.17%	
Recall	94.23%	91.09%	94.46%	95.87%	
F1 Score	95.81%	92.22%	93.81%	96.87%	
Accuracy	98.34%	97.06%	96.92%	98.11%	N = 200
Precision	97.76%	93.99%	94.11%	97.48%	
Recall	95.60%	94.09%	93.08%	94.10%	
F1 Score	96.45%	93.86%	93.36%	95.35%	

We compare the classification effects of RF classifier with different Top-N Opcodes, and the outcome is as follows, illustrated in Figure 8, where Subtrain_N stands for Top-N Opcodes extracted from the subTrain dataset and the fold line at the top portrays the variation in accuracy when the selected number of opcodes is changed. We can see that when 350, 300, 250, and 200 Opcodes are selected as the feature vector, the classification accuracy all exceed 98%, and the highest reaches up to 98.53%. We can draw the conclusion from this part of the experiments that the classification effect can meet our requirements when we use the Subtrain dataset to characterize the original Train dataset.

5.2.2. Experimental Results of Parallel Processing

This section is divided into stages of feature extraction, classifier training, and practical detection. We carry out cross-validation according to serial processing and parallel processing separately, and compare the experimental results of the two different processing modes.

Experimental Results of Parallel Processing of Train Dataset with Top-N Opcodes Extracted from Subtrain

Because the Subtrain is small, it takes only 3.86 s to extract the Opcode lists from the Subtrain subset. Then Top-N Opcodes are selected as the feature vectors to generate the feature matrix for the Train dataset, which is a time-consuming process. We use the parallel processing mode designed in Section 4.2 to realize the feature generation process. The effect of parallel processing with different parameters is shown in Table 6, where the first row, designated with a yellow background, is serial processing, and the most efficient parameter settings for parallel processing are indicated in red. We compare the experimental result of parallel processing visually and compare the optimal time performance of each parallel setting with different Top-N, as shown in Figure 9.

Table 6. Experimental result of the parallel generation of feature vectors with Top-N Opcodes extracted from Subtrain.

Num_of_Processes	Num_of_Files_ Per_Operation	Num_of_Files_ Per_Process	Time_Cost (s) N = 350	Time_Cost (s) N = 300	Time_Cost (s) N = 250	Time_Cost (s) N = 200
1	1	1	3144.43	3123.36	3094.76	2991.99
4	8	8/4 = 2	2015.38	2017.66	2011.31	2005.77
4	12	12/4 = 3	2001.01	2055.59	2009.72	1997.88
4	16	16/4 = 4	2006.25	2004.85	2005.57	2002.26
4	20	20/4 = 5	2015.54	2021.01	2013.73	2015.71
6	12	12/6 = 2	1903.09	1905.85	1899.42	1899.04
6	18	18/6 = 3	1910.87	1909.65	1906.55	1908.51
6	24	24/6 = 4	1901.23	1903.48	1899.35	1898.98
6	30	30/6 = 5	1916.27	1907.38	1912.05	1904.48
6	36	36/6 = 6	2054.21	1900.02	1908.02	1900.46
8	16	16/8 = 2	2134.08	1802.31	1804.88	1794.36
8	24	24/8 = 3	2234.32	1814.57	1811.59	1806.29
8	32	32/8 = 4	2163.13	1800.85	1805.47	1800.34
8	40	40/8 = 5	2141.56	1814.20	1810.56	1806.71
16	32	32/16 = 2	2441.05	1784.22	1786.66	1786.38
16	48	48/16 = 3	2452.15	1804.17	1785.43	1789.29
16	64	64/16 = 4	2234.93	1804.81	1780.26	1854.18
32	64	64/32 = 2	3798.22	3366.90	3362.39	3378.62
32	96	96/32 = 3	3716.72	3454.91	3457.42	3446.34

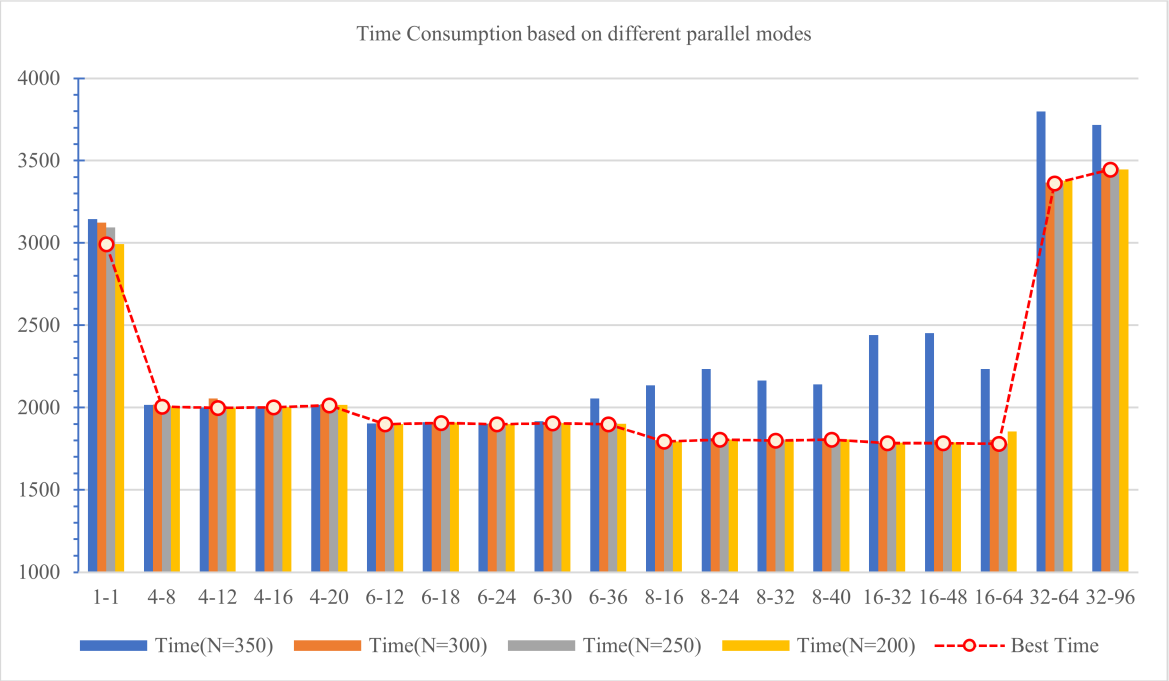


Figure 9. Comparison of time consumption based on different parallel modes.

The experimental outcomes allow us to draw the following conclusions:

(1) The smaller the Top-N that is selected, the less the processing time is, because the shorter feature vector will result in less processing workload and less time consumption.

(2) The time of parallel processing is the shortest when generating 16 processes and processing 32 samples each time, i.e., 2 samples are allocated to each process for analysis. This is because the personal PC workstation used in our experiment has eight cores. According to this setting, we can make the best use of computing resources and obtain the best results.

Experimental Results of Parallel Processing of Train dataset with Top-N Opcodes Extracted from Train

The workload will increase significantly if we extract Opcode lists directly from the Train dataset as a feature vector and then generate a feature matrix for the Train dataset. In this section, we extract feature vectors directly from the Train dataset and then carry out cross-validation. In the process of cross-validation, we extract Opcode lists from Train and construct a feature vector space for Train based on the extracted Opcode lists. Because of the large number of samples in Train dataset, these two steps are time-consuming. Therefore, we use parallel processing technology to implement these two processes, and the experimental result of parallel processing are shown in Tables 7 and 8, respectively.

Firstly, we utilized various parallel processing configurations to compile Opcode lists from the Train dataset. The findings are presented in Table 7, with the most effective parameter settings for parallel processing depicted in red, and another competitive setting depicted with a green background. A visual representation of the time-consuming comparison is illustrated in Figure 10.

Table 7. Experimental result of parallel extraction of Opcode lists from the Train dataset.

Num_of_Processes	Num_of_Files_ Per_Operation	Num_of_Files_ Per_Process	Time_Cost (s)
1	1	1	2714.30
4	8	8/4 = 2	2024.43
4	12	8/2 = 4	2006.35
4	16	10/2 = 5	1988.46
6	18	18/6 = 3	1906.77
6	24	24/6 = 4	1891.73
6	30	30/6 = 5	1890.48
6	36	36/6 = 6	1881.11
8	24	24/8 = 3	1798.20
8	32	32/8 = 4	1795.16
8	40	40/8 = 5	1798.95
16	32	32/16 = 2	1786.84
16	48	48/16 = 3	1784.62
16	64	64/16 = 4	1785.14
32	64	64/32 = 2	3377.65

Table 8. Experimental results of generating Opcode feature vectors based on parallel processing of the Train dataset.

Num_of_ Processes	Num_of_Files_ Per_Operation	Num_of_Files_ Per_Process	Time_Cost (s) N = 735	Time_Cost (s) N = 400	Time_Cost (s) N = 300	Time_Cost (s) N = 200
1	1	1	3629.03	3421.50	3094.94	2992.25
4	8	8/4 = 2	2095.58	2023.76	2018.67	2010.66
4	12	12/4 = 3	2074.93	2015.57	2000.30	2010.03
4	16	16/4 = 4	2066.56	2022.45	2016.49	2009.50
4	20	20/4 = 5	2037.21	2020.76	2038.26	2017.21
6	12	12/6 = 2	1923.39	1900.61	1903.36	1894.10
6	18	18/6 = 3	1965.54	1915.90	1911.74	1896.31
6	24	24/6 = 4	1970.55	1903.14	1899.78	1893.46
6	30	30/6 = 5	2063.14	1916.33	1910.19	1896.37
6	36	36/6 = 6	1926.12	1911.70	1898.43	1900.29
8	16	16/8 = 2	1822.72	2101.03	1800.09	1787.23
8	24	24/8 = 3	1964.03	2121.85	1809.51	1804.67
8	32	32/8 = 4	1995.37	2104.43	1792.06	1793.99
8	40	40/8 = 5	1971.58	2078.92	1811.06	1802.45
16	32	32/16 = 2	2843.86	2268.49	1782.34	1777.53
16	48	48/16 = 3	2500.37	2247.56	1775.79	1776.38
16	64	64/16 = 4	2418.35	2258.10	1771.14	1774.33
32	64	64/32 = 2	3805.67	3746.52	3360.62	3410.64
32	96	96/32 = 3	3812.21	3854.86	3461.20	3479.62

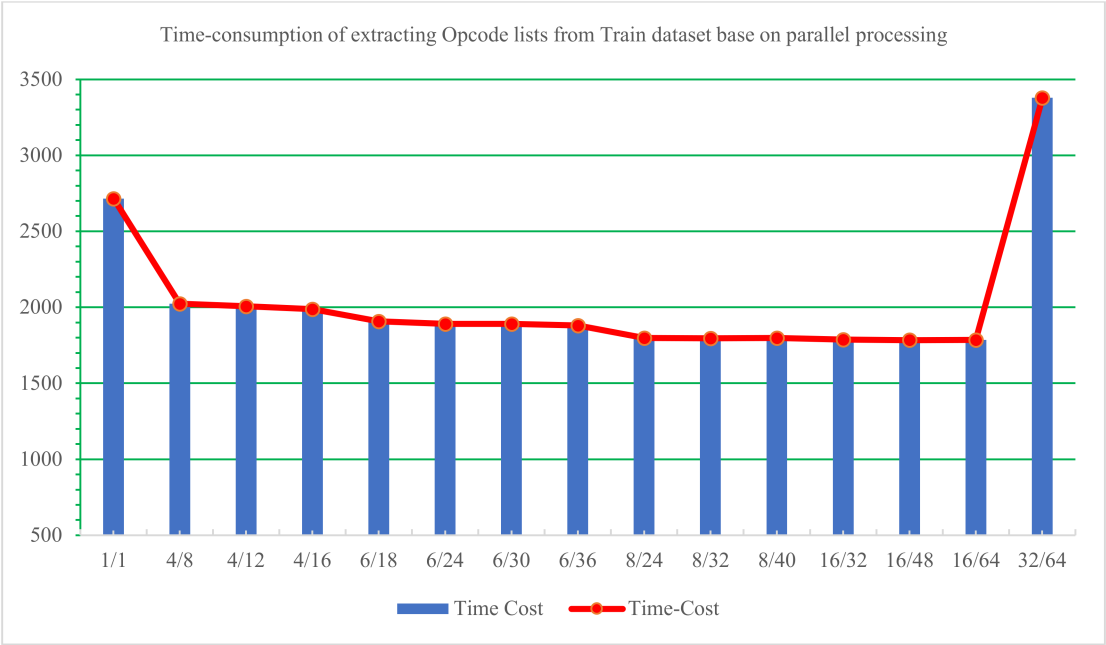


Figure 10. Time consumption of extracting Opcode lists from the Train dataset base on parallel processing.

On the basis of extracting Opcode lists from the Train dataset, the experimental results of generating feature vectors of the Train dataset based on parallel processing are shown in Table 8, with the most effective parameter settings depicted in red for each different N. The visual display of comparison of time consumption is shown in Figure 11.

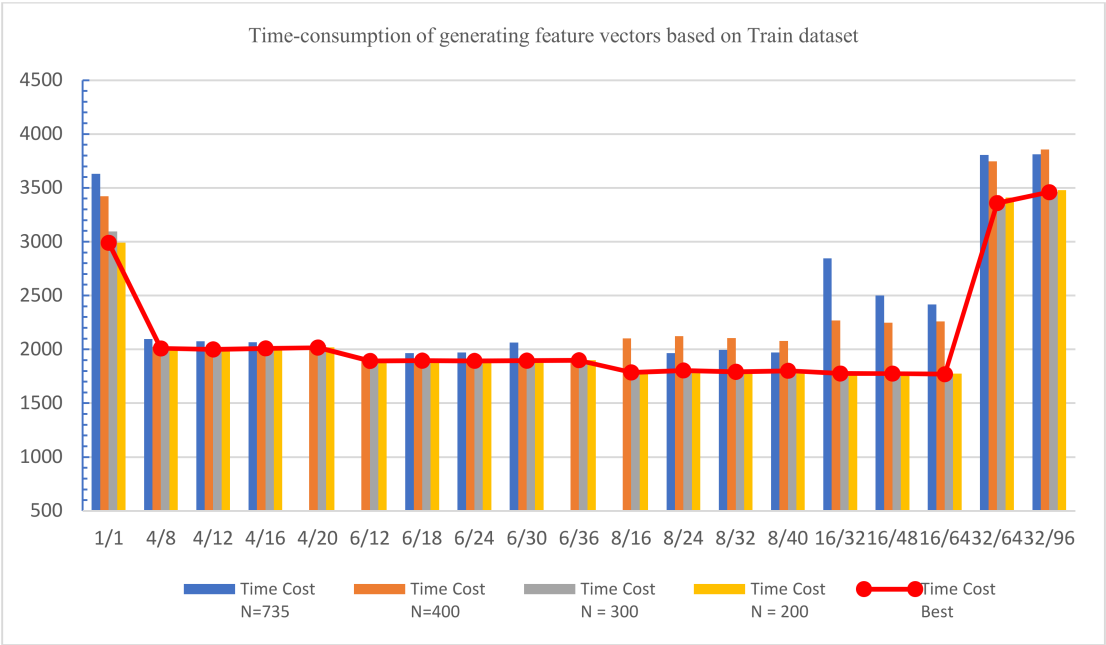


Figure 11. Time consumption of generating feature vectors based on the Train dataset.

Comparative Analysis of Overhead between Parallel Process and Serial Process

From the above experimental results of serial processing and parallel processing, we can draw the following conclusions:

(1) Parallel processing can effectively ameliorate processing efficiency

The serial processing procedures based on Train and Subtrain datasets include "extracting the feature vector from dataset → generating feature matrix for the training dataset → training classifier → practical classification". The two stages of "extracting the feature vector from dataset" and "generating feature matrix for the training dataset" are time-consuming in analyzing a massive sample set, and the application of parallel processing technology can effectively reduce the processing time.

As mentioned above, the time required to complete feature extraction and feature vector generation (Top-N = 200) based on the Subtrain subset with serial processing is 3.86 + 2991.99 = 2995.85 (s). In contrast, the optimal time consumption to complete the above process with parallel processing is 3.86 + 1786.38 = 1790.24 (s), which takes only 59.76% of the original one.

If feature extraction is based on the Train dataset, the time required to complete feature extraction and feature vector generation (Top-N = 200) is 2714.30 + 2992.25 = 5706.55 (s) with serial processing. By contrast, the optimal time consumption to complete the above process is 1784.62 + 1776.38 = 3561 (s) with parallel processing, which takes 62.4% of the original one. The parallel processing achieved promising results.

(2) Choosing the best parallel processing setting based on computing resource condition

Computing the resource platform has a fundamental impact on the parallel processing settings. As shown in the table, because our personal workstation platform has eight cores and the workload of analyzing a single malware sample is not large, the parallel processing effect is usually the best when generating 16 processes. That is to say, two processes are allocated to each core for the process, which can maximize the advantage of hardware resources. In addition, considering the allocation of computing resources on eight cores, we should not allocate too many samples to each process in parallel processing. We should ensure that each process is not overloaded to maximize the performance of each process.

5.2.3. Comparison with Similar Studies

To assess the performance of MalSEF, we compare it with related studies in terms of classification accuracy and time efficiency in this section. Since we utilize the dataset released by Microsoft in Kaggle, we select similar studies utilizing the same dataset for comparison. The four baseline models in Table 9 are classic methods that utilize the Kaggle dataset and hold significant influence, making them ideal candidates for comparison with MalSEF. The outcomes of the comparison are presented in Table 9.

In comparison to similar studies, MalSEF offers the following benefits:

(1) MalSEF strikes an optimal balance between the feature space and classification accuracy while sampling the feature vectors. When only 300 features are needed, MalSEF can achieve an accuracy of 98.53%. In contrast, the Ahmadi Mansour method [28] employed 1804 features, and the Hu Xin et al. method [29] required 2000 features, both more than six times the number of features in MalSEF. Their accuracy was only marginally higher than MalSEF by no more than 1.30%. Meanwhile, the time cost of MalSEF is significantly lower than other comparable methods. Although the classification accuracy of MalSEF may be slightly lower than the above-mentioned research, it still meets the requirements. Furthermore, when compared to other similar studies, MalSEF's feature vector is the most succinct and readily available, achieving promising classification results while minimizing the complexity of feature extraction.

Table 9. Comparison with analogous approaches.

	MalSEF	Ahmadi Mansour [28]	Hu Xin et al. [29]	Raff Edward et al. [31]	Quan Le et al. [32]
Dataset	The Microsoft Malware Classification Challenge dataset in Kaggle				
# Features	300	1804	2000	--	10,000
Feature Set	Top-N opcode list	features extracted from hex dumps + features extracted decompiled files	multifaceted content features + threat intelligence	Not clearly stated	One dimensional representation of the malware sample
Classification Accuracy	98.53%	99.77%	99.80%	97.80%	98.20%
Time cost (s)	1790.24	5656.00	2867.00	32,087.40	6372.00 (Train time for deep learning network)
Required hardware platform	Lenovo ThinkStation, Intel® Core™ i7-6700U CPU @3.40GHz × 8, 8 GB memory	A laptop with a quad-core processor (2 GHz), and 8 GB RAM	Not clearly stated	A workstation with an Intel Xeon E5-2650 CPU at 2.30 GHz, 128 GB of RAM, and 4 TB of SSD storage	A workstation with a 6 core i7-6850K Intel processor

Features means the number of Features.

(2) MalSEF really realized the parallel analysis of massive malware, which can effectively reduce the analysis time. Compared with the serial processing, the time efficiency is improved by 37.60%. Compared with similar research, the handing time required by MalSEF is the shortest. There may be concerns over whether the baselines could achieve better time efficiency if they use a smaller feature set. The aforementioned comparison of models is conducted under the terms of their respective required feature size. When only 300 features are used, MalSEF can achieve ideal detection results. However, if the size of the features in the baseline modes decreases, it is not confirmed whether their accuracy can remain. This raises a promising question for future research. It can also be observed that the time reduction is not especially significant when the feature size is decreased. This is because the dataset is not particularly large, and as the size of the dataset increases, the time efficiency of MalSEF will become more apparent.

(3) The performance of the hardware platform required by MalSEF is moderate, so ordinary researchers can earn the opportunity. As a consequence, it can be generalized in the field of popular network security and has good applicability.

(4) As for the similarities among variants of malware, MalSEF provides and verifies semantic explanations by extracting and mining Opcode information from malware samples, which compensates for the lack of semantic explanations in deep learning-based malware classification.

6. Conclusions

Considering the widespread use of metamorphic techniques by malware to evade detection, we conducted a comparative analysis of the similarities between malware variants. We found that variants are reproduced in the form of filling old wine in a new bottle actually, and there are manifest similarities between variants. To enhance the efficiency of malware analysis, accurately classifying these variants into their families is of theoretical and practical value. To achieve this goal, we proposed a new malware classification framework called MalSEF. MalSEF adopts a lightweight feature engineering framework by extracting a small part of the samples from the larger dataset to construct a sampled subset. Based on the sampled subset, it generates feature vectors to represent the original samples dataset, and then generates lightweight feature matrices, thereby reducing the workload of

generating feature matrices from a large number of samples. Based on the above theory, our method uses multi-core parallel processing to analyze malware, thus making full use of the computing resources of the modern personal PC. Compared with the traditional serial processing, the time consumption is decreased and the analysis efficiency is promoted significantly. The key advantage of this paper is that MalSEF provides a practical and effective approach for analyzing and processing vast amounts of malware on personal computers. Going forward, we aim to explore the following areas: (1) Since cloud computing platforms are extensively utilized in the realm of network security, we can leverage these resources to conduct extensive parallel processing [38]; (2) utilizing Opcode to depict malware's features is a practical approach. By doing so, we can conduct a multi-dimensional malware analysis and utilize various features for a more comprehensive portrayal of malware [39]. (3) With the increasing frequency of APT attacks and the large-scale malware outbreaks in APT, how to apply parallel processing technology to detect APT attacks and take timely response measures is also a valuable research direction in the future [40,41].

Author Contributions: Conceptualization, W.H.; Methodology, J.L.; Software, W.H.; Validation, J.L.; Investigation, Q.Z.; Data curation, Y.Z.; Writing—original draft, W.H.; Writing—review & editing, J.L. and Q.Z.; Supervision, J.X.; Project administration, J.X.; Funding acquisition, J.X. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the National Natural Science Foundation of China (No. 62172042), the National Key Research & Development Program of China (2020YFB1712104), and the Major Scientific and Technological Innovation Projects of Shandong Province (2020CXGC010116).

Data Availability Statement: The data used to validate the outcomes of this investigation can be obtained by contacting the corresponding author.

Conflicts of Interest: There is no potential conflict of interest in the submission of this manuscript, and all authors have approved its publication.

References

1. Rezaei, T.; Manavi, F.; Hamzeh, A. A PE header-based method for malware detection using clustering and deep embedding techniques. *J. Inf. Secur. Appl.* **2021**, *60*, 102876. [CrossRef]
2. Darem, A.; Abawajy, J.; Makkar, A.; Alhashmi, A.; Alanazi, S. Visualization and deep-learning-based malware variant detection using OpCode-level features. *Future Gener. Comput. Syst.* **2021**, *125*, 314–323. [CrossRef]
3. Malware. Available online: https://www.av-test.org/en/statistics/malware/ (accessed on 7 July 2021).
4. Singh, J.; Singh, J. A survey on machine learning-based malware detection in executable files. *J. Syst. Archit.* **2021**, *112*, 101861. [CrossRef]
5. Botacin, M.; Ceschin, F.; Sun, R.; Oliveira, D.; Grégio, A. Challenges and pitfalls in malware research. *Comput. Secur.* **2021**, *106*, 102287. [CrossRef]
6. Han, W.; Xue, J.; Wang, Y.; Huang, L.; Kong, Z.; Mao, L. MalDAE: Detecting and explaining malware based on correlation and fusion of static and dynamic characteristics. *Comput. Secur.* **2019**, *83*, 208–233. [CrossRef]
7. Santos, I.; Brezo, F.; Ugarte-Pedrero, X.; Bringas, P.G. Opcode sequences as representation of executables for data-mining-based unknown malware detection. *Inf. Sci.* **2013**, *231*, 64–82. [CrossRef]
8. Tien, C.W.; Chen, S.W.; Ban, T.; Kuo, S.Y. Machine learning framework to analyze IoT malware using elf and opcode features. *Digit. Threat. Res. Pract.* **2020**, *1*, 5. [CrossRef]
9. Ling, Y.T.; Sani, N.F.M.; Abdullah, M.T.; Hamid, N.A.W.A. Structural features with nonnegative matrix factorization for metamorphic malware detection. *Comput. Secur.* **2021**, *104*, 102216. [CrossRef]
10. Zheng, J.; Zhang, Y.; Li, Y.; Wu, S.; Yu, X. Towards Evaluating the Robustness of Adversarial Attacks Against Image Scaling Transformation. *Chin. J. Electron.* **2023**, *32*, 151–158. [CrossRef]
11. Zhang, Q.; Ma, W.; Wang, Y.; Zhang, Y.; Shi, Z.; Li, Y. Backdoor attacks on image classification models in deep neural networks. *Chin. J. Electron.* **2022**, *31*, 199–212. [CrossRef]
12. Guo, F.; Zhao, Q.; Li, X.; Kuang, X.; Zhang, J.; Han, Y.; Tan, Y.A. Detecting adversarial examples via prediction difference for deep neural networks. *Inf. Sci.* **2019**, *501*, 182–192. [CrossRef]
13. Rudd, E.M.; Rozsa, A.; Günther, M.; Boult, T.E. A survey of stealth malware attacks, mitigation measures, and steps toward autonomous open world solutions. *IEEE Commun. Surv. Tutor.* **2016**, *19*, 1145–1172. [CrossRef]
14. Microsoft Malware Classification Challenge, Kaggle. Available online: https://www.kaggle.com/c/malware-classification (accessed on 27 October 2022).

15. Radkani, E.; Hashemi, S.; Keshavarz-Haddad, A.; Amir Haeri, M. An entropy-based distance measure for analyzing and detecting metamorphic malware. *Appl. Intell.* **2018**, *48*, 1536–1546. [CrossRef]
16. Yagemann, C.; Sultana, S.; Chen, L.; Lee, W. *Barnum*: Detecting document malware via control flow anomalies in hardware traces. In Proceedings of the Information Security: 22nd International Conference, ISC 2019, New York City, NY, USA, 16–18 September 2019; Proceedings 22. Springer International Publishing: Cham, Switzerland, 2019; pp. 341–359.
17. Ye, Y.; Li, T.; Adjeroh, D.; Iyengar, S.S. A survey on malware detection using data mining techniques. *ACM Comput. Surv. (CSUR)* **2017**, *50*, 1–40. [CrossRef]
18. Fan, Y.; Ye, Y.; Chen, L. Malicious sequential pattern mining for automatic malware detection. *Expert Syst. Appl.* **2016**, *52*, 16–25. [CrossRef]
19. Burnap, P.; French, R.; Turner, F.; Jones, K. Malware classification using self organising feature maps and machine activity data. *Comput. Secur.* **2018**, *73*, 399–410. [CrossRef]
20. Garcia, D.E.; DeCastro-Garcia, N. Optimal feature configuration for dynamic malware detection. *Comput. Secur.* **2021**, *105*, 102250. [CrossRef]
21. Han, W.; Xue, J.; Wang, Y.; Liu, Z.; Kong, Z. MalInsight: A systematic profiling based malware detection framework. *J. Netw. Comput. Appl.* **2019**, *125*, 236–250. [CrossRef]
22. Guerra-Manzanares, A.; Bahsi, H.; Nõmm, S. Kronodroid: Time-based hybrid-featured dataset for effective android malware detection and characterization. *Comput. Secur.* **2021**, *110*, 102399. [CrossRef]
23. Xin, Y.; Xie, Z.Q.; Yang, J. A load balance oriented cost efficient scheduling method for parallel tasks. *J. Netw. Comput. Appl.* **2017**, *81*, 37–46. [CrossRef]
24. Smilovich, D.; Radovitzky, R.; Dvorkin, E. A parallel staggered hydraulic fracture simulator incorporating fluid lag. *Comput. Methods Appl. Mech. Eng.* **2021**, *384*, 114003. [CrossRef]
25. Wang, K.; Li, X.; Gao, L.; Li, P.; Gupta, S.M. A genetic simulated annealing algorithm for parallel partial disassembly line balancing problem. *Appl. Soft Comput.* **2021**, *107*, 107404. [CrossRef]
26. Bailey, M.; Oberheide, J.; Andersen, J.; Mao, Z.M.; Jahanian, F.; Nazario, J. Automated classification and analysis of internet malware. In Proceedings of the Recent Advances in Intrusion Detection: 10th International Symposium, RAID 2007, Gold Goast, Australia, 5–7 September 2007; Proceedings 10. Springer: Berlin/Heidelberg, Germany, 2007; pp. 178–197.
27. Nataraj, L.; Yegneswaran, V.; Porras, P.; Zhang, J. A comparative assessment of malware classification using binary texture analysis and dynamic analysis. In Proceedings of the 4th ACM Workshop on Security and Artificial Intelligence, Chicago, IL, USA, 21 October 2011; pp. 21–30.
28. Ahmadi, M.; Ulyanov, D.; Semenov, S.; Trofimov, M.; Giacinto, G. Novel feature extraction, selection and fusion for effective malware family classification. In Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy, New Orleans, LA, USA, 9–11 March 2016; pp. 183–194.
29. Hu, X.; Jang, J.; Wang, T.; Ashraf, Z.; Stoecklin, M.P.; Kirat, D. Scalable malware classification with multifaceted content features and threat intelligence. *IBM J. Res. Dev.* **2016**, *60*, 6:1–6:11. [CrossRef]
30. Lee, T.; Kwak, J. Effective and reliable malware group classification for a massive malware environment. *Int. J. Distrib. Sens. Netw.* **2016**, *12*, 4601847. [CrossRef]
31. Raff, E.; Nicholas, C. Malware classification and class imbalance via stochastic hashed lzjd. In Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security, Dallas, TX, USA, 3 November 2017; pp. 111–120.
32. Le, Q.; Boydell, O.; Mac Namee, B.; Scanlon, M. Deep learning at the shallow end: Malware classification for non-domain experts. *Digit. Investig.* **2018**, *26*, S118–S126. [CrossRef]
33. Nakazato, J.; Song, J.; Eto, M.; Inoue, D.; Nakao, K. A novel malware clustering method using frequency of function call traces in parallel threads. *IEICE Trans. Inf. Syst.* **2011**, *94*, 2150–2158. [CrossRef]
34. Sheen, S.; Anitha, R.; Sirisha, P. Malware detection by pruning of parallel ensembles using harmony search. *Pattern Recognit. Lett.* **2013**, *34*, 1679–1686. [CrossRef]
35. Wang, X.; Zhang, D.; Su, X.; Li, W. Mlifdect: Android malware detection based on parallel machine learning and information fusion. *Secur. Commun. Netw.* **2017**, *2017*, 6451260. [CrossRef]
36. Kabir, E.; Hu, J.; Wang, H.; Zhuo, G. A novel statistical technique for intrusion detection systems. *Future Gener. Comput. Syst.* **2018**, *79*, 303–318. [CrossRef]
37. Abusitta, A.; Li, M.Q.; Fung, B.C. Malware classification and composition analysis: A survey of recent developments. *J. Inf. Secur. Appl.* **2021**, *59*, 102828. [CrossRef]
38. Mishra, P.; Verma, I.; Gupta, S. KVMInspector: KVM Based introspection approach to detect malware in cloud environment. *J. Inf. Secur. Appl.* **2020**, *51*, 102460. [CrossRef]
39. Wang, P.; Tang, Z.; Wang, J. A novel few-shot malware classification approach for unknown family recognition with multi-prototype modeling. *Comput. Secur.* **2021**, *106*, 102273. [CrossRef]

40. Han, W.; Xue, J.; Wang, Y.; Zhang, F.; Gao, X. APTMalInsight: Identify and cognize APT malware based on system call information and ontology knowledge framework. *Inf. Sci.* **2021**, *546*, 633–664. [CrossRef]
41. Liras, L.F.M.; de Soto, A.R.; Prada, M.A. Feature analysis for data-driven APT-related malware discrimination. *Comput. Secur.* **2021**, *104*, 102202. [CrossRef]

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