Performance Comparison of Training Datasets for System Call-Based Malware Detection with Thread Information

Yuki KAJIWARA\(^{1,\ast}a\), Nonmember, Junjun ZHENG\(^{1,b}\), and Koichi MOURI\(^{1,c}\), Members

SUMMARY The number of malware, including variants and new types, is dramatically increasing over the years, posing one of the greatest cybersecurity threats nowadays. To counteract such security threats, it is crucial to detect malware accurately and early enough. The recent advances in machine learning technology have brought increasing interest in malware detection. A number of research studies have been conducted in the field. It is well known that malware detection accuracy largely depends on the training dataset used. Creating a suitable training dataset for efficient malware detection is thus crucial. Different works usually use their own dataset; therefore, a dataset is only effective for one detection method, and strictly comparing several methods using a common training dataset is difficult. In this paper, we focus on how to create a training dataset for efficiently detecting malware. To achieve our goal, the first step is to clarify the information that can accurately characterize malware. This paper concentrates on threads, by treating them as important information for characterizing malware. Specifically, on the basis of the dynamic analysis log from the Alkanet, a system call tracer, we obtain the thread information and classify the thread information processing into four patterns. Then the malware detection is performed using the number of transitions of system calls appearing in the thread as a feature. Our comparative experimental results showed that the primary thread information is important and useful for detecting malware with high accuracy.

key words: malware detection, machine learning, system calls, thread

1. Introduction

Lately, a large number of unknown malware and variants are discovered in addition to existing malware. Malware spread has posed severe security issues on cyber systems\(^{[1, 2]}\). To face malware threats, it is important to accurately detect malware and take proper actions, such as damage prevention at an early stage. Benefiting from the rapid development of machine learning, a large number of malware detection methods based on machine learning have been proposed\(^{[3, 4]}\). Machine learning is a technology that causes a computer to learn various data and predict and estimate the correct label for learning. In unsupervised learning, no labels are given in advance to the algorithm. The fundamental idea of reinforcement learning is to select an optimal strategy that maximizes the expected reward of agents. Machine learning helps improve malware detection accuracy. It is a highly promising approach for malware detection and classification.

In machine learning-based malware detection, training features are generated based on the malware datasets and general executable files that do not demonstrate any malicious behavior (hereinafter referred to as cleanware). This means that the detection accuracy largely depends on the training dataset used. Creating a suitable training dataset for efficient malware detection is crucial. In literature, few studies considered multiple datasets for one proposed method. In other words, different studies usually use their own dataset. This makes a dataset effective only for a specific detection method and makes comparisons with a common training dataset extremely difficult. To compare features and machine learning-based detection methods with existing studies, it is necessary to use the same dataset. Using the openly provided dataset, such as those provided by malware workshops (MWS)\(^{[6]}\), makes it possible to prepare the same dataset for training. However, what makes it hard to compare different methods using the same dataset is the fact that researchers usually have to collect data by themselves from various sources such as websites or companies. As a result, the use of different datasets makes it impossible to strictly compare the proposed method with existing studies. Hence, it becomes difficult to judge whether the proposed method can be effectively applied accurately.

To date, these commonly used malware datasets, including FFRI datasets, which are also part of the MWS datasets, have been typically acquired by using dynamic analysis environments such as Cuckoo Sandbox\(^{[7]}\) and Alkanet\(^{[8, 9]}\). Conversely, cleanware datasets were generally collected from free download portals, such as Cnet\(^{[10]}\) and Softpedia\(^{[11]}\) by individuals\(^{[10, 11]}\). In general, malware detection was performed by machine learning using system call-based features\(^{[12]}\). However, how to create an appropriate training dataset for efficient malware detection, particularly which information should be concentrated and regarded as features, still remains unclear.

In this paper, we focus on creating training datasets us-

---

\(^{1}\)The authors are with the College of Information Science and Engineering, Ritsumeikan University, Kusatsu-shi, 525–8577 Japan.

\(^{\ast}\)Presently with NEC Corporation, Tokyo, 108–8001 Japan.

a) E-mail: ykajiwara@asl.cs.ritsumei.ac.jp
b) E-mail: jzheng@asl.cs.ritsumei.ac.jp
c) E-mail: mouri@cs.ritsumei.ac.jp

DOI: 10.1587/transinf.2021EDP7067

---

Copyright © 2021 The Institute of Electronics, Information and Communication Engineers
ing thread information, with the aim to (i) efficiently detect malware and (ii) reveal the information that can characterize the dynamic behavior of malware, thereby providing insight into how to create an effective dataset for malware training. Specifically, we use the dynamic analysis log from the Alkanet, a system call tracer [9], to obtain the thread information. Then the processing of these extracted thread information is classified into four patterns, on the basis of whether these threads are primary ones or not, for generating four types of training datasets. In particular, malware detection is performed using the number of system call transitions appearing in the threads as a feature. In numerical experiments, we validate the created datasets and compare them in terms of detection accuracy. The main contributions of this work are twofold:

- providing new insight on how to create an effective dataset for machine learning-based malware detection using thread information and
- clarifying the important information for accurately characterizing malware dynamics.

The rest of the paper is organized as follows. Section 2 introduces related research works. In Sect. 3, we present a system call-based malware detection approach using thread information. In particular, four types of training datasets are created according to different thread processing methods. Then, comparative experiments using these created training datasets are described in Sect. 4, and their results are discussed in Sect. 5. Finally, Sect. 6 summarizes this work with some remarks and introduces the future directions.

2. Related Works

Machine learning has been widely applied in the field of malware detection over the past decade. For example, Garg et al. [13] detected malware using multiple supervised machine learning models, using the frequency of application programming interface (API) calls as a feature. They focused on the samples that invoke about 100 to 700 API calls and calculated the API call frequency on the basis of the records of API calls via IDA-Pro [14] during static analysis. Six types of supervised learning models, namely, k-nearest neighbor (kNN), linear discriminant analysis (LDA), decision tree (DT), random forest (RF), neural network (NN), and support vector machine (SVM), were considered for malware detection using the API call frequency. The detection accuracy provided by SVM became the highest, approximately 93%.

Ahsan-Ul-Haque et al. [15] proposed an approach that uses system calls for detecting malware on Android devices. They analyzed the sequence of system call logs that are made by malware and benign applications and extracted a set of distinctive features by using a Markov model, considering the probability that a certain system call appears one after another. System call information for each application was collected using the standard Linux debug utility strace. To decrease the training time, they proposed a feature reduction method using Gaussian dissimilarity. To demonstrate the effectiveness of the proposed method to detect malware, they used a simple machine learning model called the Gaussian Bayes classifier. Using two malware datasets and one benign dataset, they evaluated the model using 10-fold cross-validation and achieved 98% accuracy.

Xiao et al. [16] presented a method that uses the transition probability of system calls in Markov chains for detecting Android malware. The system call sequences were described by stationary Markov chains, and a backpropagation NN was applied to detect malware based on Markov chains. In particular, assuming that the transition probability from one system call to another is significantly different between malware and benign applications, the malware was detected by comparing the transition probabilities in the Markov chain and capturing the abnormal state transition.

Darshan et al. [17] introduced a system call-based Windows malware detection method using Cuckoo Sandbox logs. After converting the JSON format log into MIST format, the features of n-grams were generated based on the value indicating the system call. Performance evaluation was performed using six types of learning models, namely, Bayesian-Logistic-Regression, SPegasos, IB1, Bagging, Part, and J48, using the WEKA tool. Their work has shown that SPegasos had the best performance in terms of accuracy, true-positive rate (TPR), and false-positive rate (FPR).

Moreover, Woźniak et al. [18] proposed a model that uses optimized Internet of Thing (IoT) network information to detect malicious networking threads in connecting Android system-based IoT cyber-physical devices to the network. Malware detection was based on analyzing malicious traffic based on external IPs and TCP packets using a long-short-term-memory (LSTM)-based recurrent neural network (RNN) classifier.

In this paper, malware detection is based on the system calls that are obtained from the dynamic analysis logs of Windows executable files. In particular, we use Alkanet [9], a dynamic analysis environment that can comprehensively acquire thread-level information. On the other hand, although these existing works considered the malware detection or classification problems, there is still a lack of knowledge on how to create an effective training dataset for malware detection or classification, in other words, the question of how to explore and evaluate a dataset creation method keeps unclear. Thus this paper aims to solve the above challenges.

3. System Call-Based Malware Detection Using Thread Information

In this section, we first describe the log of dynamic analysis and then discuss which information could characterize the malware in terms of both system calls and threads. Next, we present the methods of creating four types of training datasets using the extracted information from the log. Fi-
nally, we introduce the feature generating from four types of datasets for learning, the used learning method, and performance measures for malware detection.

3.1 Alkanet Log

It is well known that although static malware analysis is quick and easy, it fails to detect advanced malware and capture important dynamic behaviors. Dynamic analysis could provide more information on malware than static analysis. Related studies [10], [19] showed that the malware detection using the log of dynamic analysis could effectively avoid the impact of obfuscation. Therefore, this paper considers the dynamic analysis that runs the malware on a virtualized environment and traces its dynamics from the invocation of system calls.

Alkanet [9], a dynamic analysis environment, which was developed in our lab and proved to be an effective route to understanding the actual malware behaviors [8], is taken into account. Figure 1 illustrates the architecture of the Alkanet implemented on Windows XP x86 32-bit operating system. The system is composed of two computers; one is malware observation PC, which executes a malware and hooks the system calls invoked by the malware, and another is logging PC, which records and analyzes the trace log of malware execution. More specifically, Alkanet operates as an extension of BitVisor [20], which is a virtual machine monitor (VMM) and records the log of system calls invoked by processes on the guest OS. Since the entry point of the system call exists in the kernel space, it cannot be accessed from the process operating in the user mode. Furthermore, the system call cannot be called by avoiding the entry point. To our knowledge, as long as the malware runs in user mode, it has to

| No. | Description     |
|-----|-----------------|
| 7   | Log serial number|
| 8   | Log serial number |
| 9   | Log serial number |
| 10  | Log serial number |
| 11  | Log serial number |
| 12  | Log serial number |
| 13  | Log serial number |
| 14  | Log serial number |
| 15  | Log serial number |
| 16  | Log serial number |
| 17  | Log serial number |
| 18  | Log serial number |
| 19  | Log serial number |
| 20  | Log serial number |
| 21  | Log serial number |
| 22  | Log serial number |
| 23  | Log serial number |
| 24  | Log serial number |
| 25  | Log serial number |
| 26  | Log serial number |
| 27  | Log serial number |
| 28  | Log serial number |
| 29  | Log serial number |
| 30  | Log serial number |
| 31  | Log serial number |
| 32  | Log serial number |
| 33  | Log serial number |
| 34  | Log serial number |
| 35  | Log serial number |
| 36  | Log serial number |
| 37  | Log serial number |
| 38  | Log serial number |
| 39  | Log serial number |
| 40  | Log serial number |
| 41  | Log serial number |
| 42  | Log serial number |
| 43  | Log serial number |
| 44  | Log serial number |
| 45  | Log serial number |
| 46  | Log serial number |
| 47  | Log serial number |
| 48  | Log serial number |
| 49  | Log serial number |
| 50  | Log serial number |
| 51  | Log serial number |
| 52  | Log serial number |
| 53  | Log serial number |
| 54  | Log serial number |
| 55  | Log serial number |
| 56  | Log serial number |
| 57  | Log serial number |
| 58  | Log serial number |
| 59  | Log serial number |
| 60  | Log serial number |
| 61  | Log serial number |
| 62  | Log serial number |
| 63  | Log serial number |
| 64  | Log serial number |
| 65  | Log serial number |
| 66  | Log serial number |
| 67  | Log serial number |
| 68  | Log serial number |
| 69  | Log serial number |
| 70  | Log serial number |
| 71  | Log serial number |
| 72  | Log serial number |
| 73  | Log serial number |
| 74  | Log serial number |
| 75  | Log serial number |
| 76  | Log serial number |
| 77  | Log serial number |
| 78  | Log serial number |
| 79  | Log serial number |
| 80  | Log serial number |
| 81  | Log serial number |
| 82  | Log serial number |
| 83  | Log serial number |
| 84  | Log serial number |
| 85  | Log serial number |
| 86  | Log serial number |
| 87  | Log serial number |
| 88  | Log serial number |
| 89  | Log serial number |
| 90  | Log serial number |
| 91  | Log serial number |
| 92  | Log serial number |
| 93  | Log serial number |
| 94  | Log serial number |
| 95  | Log serial number |
| 96  | Log serial number |
| 97  | Log serial number |
| 98  | Log serial number |
| 99  | Log serial number |
| 100 | Log serial number |
| 101 | Log serial number |
| 102 | Log serial number |
| 103 | Log serial number |
| 104 | Log serial number |
| 105 | Log serial number |
| 106 | Log serial number |
| 107 | Log serial number |
| 108 | Log serial number |
| 109 | Log serial number |
| 110 | Log serial number |
| 111 | Log serial number |
| 112 | Log serial number |
| 113 | Log serial number |
| 114 | Log serial number |
| 115 | Log serial number |
| 116 | Log serial number |
| 117 | Log serial number |
| 118 | Log serial number |
| 119 | Log serial number |
| 120 | Log serial number |
| 121 | Log serial number |
| 122 | Log serial number |
| 123 | Log serial number |
| 124 | Log serial number |
| 125 | Log serial number |
| 126 | Log serial number |
| 127 | Log serial number |
| 128 | Log serial number |
| 129 | Log serial number |
| 130 | Log serial number |
| 131 | Log serial number |
| 132 | Log serial number |
| 133 | Log serial number |
| 134 | Log serial number |
| 135 | Log serial number |
| 136 | Log serial number |
| 137 | Log serial number |
| 138 | Log serial number |
| 139 | Log serial number |
| 140 | Log serial number |
| 141 | Log serial number |
| 142 | Log serial number |
| 143 | Log serial number |
| 144 | Log serial number |
| 145 | Log serial number |
| 146 | Log serial number |
| 147 | Log serial number |
| 148 | Log serial number |
| 149 | Log serial number |
| 150 | Log serial number |
| 151 | Log serial number |
| 152 | Log serial number |
| 153 | Log serial number |
| 154 | Log serial number |
| 155 | Log serial number |
| 156 | Log serial number |
| 157 | Log serial number |
| 158 | Log serial number |
| 159 | Log serial number |
| 160 | Log serial number |
| 161 | Log serial number |
| 162 | Log serial number |
| 163 | Log serial number |
| 164 | Log serial number |
| 165 | Log serial number |
| 166 | Log serial number |
| 167 | Log serial number |
| 168 | Log serial number |
| 169 | Log serial number |
| 170 | Log serial number |
| 171 | Log serial number |
| 172 | Log serial number |
| 173 | Log serial number |
| 174 | Log serial number |
| 175 | Log serial number |
| 176 | Log serial number |
| 177 | Log serial number |
| 178 | Log serial number |
| 179 | Log serial number |
| 180 | Log serial number |
| 181 | Log serial number |
| 182 | Log serial number |
| 183 | Log serial number |
| 184 | Log serial number |
| 185 | Log serial number |
| 186 | Log serial number |
| 187 | Log serial number |
| 188 | Log serial number |
| 189 | Log serial number |
| 190 | Log serial number |
| 191 | Log serial number |
| 192 | Log serial number |
| 193 | Log serial number |
| 194 | Log serial number |
| 195 | Log serial number |
| 196 | Log serial number |
| 197 | Log serial number |
| 198 | Log serial number |
| 199 | Log serial number |
| 200 | Log serial number |

Figure 2 depicts an example of the dynamic analysis log of Alkanet, whose details are summarized in Table 1. Alkanet records two logs; one is for `sysenter` and the other is for `sysexit`, in response to the issuance of a system call. In this work, the filtering is performed in such a way that only the system calls at the time of `sysenter` are extracted. The system call sequence is essentially a time-series data that is finally generated.

3.2 System Call Sequence

When using a dynamic analysis log for malware detection, it is desirable to use a log that provides information that helps better understand the actual malware behavior timely. In this paper, we treat the system call trace log as a dynamic analysis log.

An important objective of the dynamic analysis is to observe system calls invoked by malware. In dynamic analysis, the smallest unit of observable behavior is a machine language instruction unit or a memory access unit, which enables a detailed analysis of the malware behavior. However, the fine granularity of the above units makes malware behavior analysis time-consuming, and the analysis overhead becomes large.

The system call trace records the invoked system calls during malware execution, and the hook of system calls occurs only when they are invoked by the malware, which means a small overhead. Furthermore, since the system call entry point exists in the kernel space, it cannot be accessed by a process running in the user mode. Malware cannot intentionally avoid system call hooks because it cannot be called without going through the entry point. To our knowledge, as long as the malware runs in user mode, it has to
invoke system calls to affect the system by performing file operations and communication. The system call trace log is considered an effective route to grasping malware behavior with a low overhead. In this paper, from the Alkanet log, the system calls that appear in the thread generated during the execution of each malware sample are arranged in the order of issuance time and are used to represent malware behavior.

3.3 Threads

Related works on malware detection often take into account opcode information[21] and system or API call information[22],[23]. However, a small number of studies focus on other information such as processes[24] and threads[25], which are also practically important. When the malware is executed, it is, in fact, the threads that are actually executed, and it is thought that malware behavior appears more clearly in threads than in processes. In this paper, we focus on threads for malware characterization. In particular, among the threads created during malware execution, the characteristic behavior of malware often appears in the primary thread. Nevertheless, the threads other than the primary thread should not be completely ignored, since the behavior that characterizes malware may appear in the threads following the primary one.

To attack the target system, a malicious thread needs to invoke a system call. However, the threads generated by malicious code and on which ones should receive focus are unknown. Thus, it is important to distinguish the system call trace log from the thread-level point of view when performing system call-based malware detection. We also call it thread-based malware detection, where the thread processing is classified into four patterns, on the basis of whether these threads are primary or not, for generating four types of training datasets, which are described in the following subsection.

3.4 Four Types of Training Datasets

In this paper, we created training data from the system call trace logs of both malware and cleanware samples acquired from the Alkanet dynamic analysis. As mentioned in Sect. 3.3, thread processing (i.e., the methods of creating a system call sequence) generated during sample execution was classified into the following four patterns:

**Pattern 1**: a system call sequence in which all system calls appeared only in the primary threads;

**Pattern 2**: a system call sequence in which the system calls were extracted from all threads;

**Pattern 3**: a system call sequence in which all system calls were extracted from non-primary threads; and

**Pattern 4**: a system call sequence in which all system calls were extracted from every single thread.

It should be noticed that the first three create system call sequences focus on samples, whereas the last one just focuses on the system call sequence for each thread during sample execution. According to the above patterns, four types of training datasets were obtained. Figure 3 illustrates the target threads for creating system call sequences as training data for a sample. In this figure, Thread 1 represents the primary thread and Thread i (i = 2, ..., n) refers to a non-primary thread such as a child thread. Besides, the colored box corresponds to the target thread(s) of a sample for creating the system call sequence for learning. In general, characteristic behaviors, for example, the packer processing, usually appear in the primary thread. It should also be noted that some malware-like behavior may appear in the threads that follow the primary thread. Considering both primary and non-primary threads together would improve malware detection. To investigate which kind of threads is more valuable, we consider Patterns 1, 2, and 3.

In addition, the thread generated during sample execution does not always behave at the same level as the sample. Specifically, when looking at the threads individually, the threads generated during the execution of the malware sample could behave like cleanware, and vice versa. Therefore, Pattern 4 is presented to investigate how to detect a malware based on the detection results of its generated threads.

3.5 Feature Generation and Learning Method

The feature of interest in this paper is the number of system call transitions that appeared in the thread generated during each sample execution. The learning method we adopted is RF[26], a popular learning method combined with a series of DT classifiers, owing to its high accuracy, and outlier tolerance, and considering that it does not cause overfitting problems.

Malware needs to invoke system calls to attack the system so that it is intuitively believed that the malware behavior is reflected in the system call issuance order and the number of issuances. In addition, information regarding the appearance and transition of system calls in the log was also used as a feature in existing research works. A large number of them demonstrated high detection accuracy[10],[12],[16]. For example, in our previous work[12], we conducted an experiment using four features
and three machine learning algorithms to compare the detection accuracies of 12 kinds of features combined with the learning algorithms. Specifically, we created learning data with four features (the number of appearances, the probability of appearance, the number of transitions, and the transition probability) and compared three machine learning algorithms (RF, SVM, and logistic regression). The accuracy of malware detection by RF, which used the number of system call transitions as a feature, proved to be the highest. Therefore, we utilize this combination in this paper.

Besides, the system calls targeted for feature generation are all system calls that appear in the system call log for both malware and cleanware samples used. For brevity, a total of 176 kinds of system calls have been observed, and the number of transition patterns among two system calls are 30,976 (= 176 × 176).

3.6 Performance Measures

Cross-validation is an efficient way to evaluate the performance of machine learning models. In k-fold cross-validation, a special case of cross-validation, data is divided into k parts; (k − 1) data are used as training data, and the remaining one is used for verification. The model performance is measured k times, and the final performance is given by the average value of k times. In this paper, we consider the 10-fold cross-validation and evaluate the detection results by using a confusion matrix as shown in Table 2. The confusion matrix represents the actual class in the vertical axis and the class predicted by the model in the horizontal axis. In this table, two classes mean malware and cleanware, and each element is described as follows:

\[
\begin{array}{c|cc}
\text{Real} \backslash \text{Prediction} & \text{Malware} & \text{Cleanware} \\
\hline
\text{Malware} & \text{TP} & \text{FN} \\
\text{(True Positive)} & \text{(False Negative)} \\
\hline
\text{Cleanware} & \text{FP} & \text{TN} \\
\text{(False Positive)} & \text{(True Negative)}
\end{array}
\]

Table 2 Confusion matrix.

and three machine learning algorithms to compare the detection accuracies of 12 kinds of features combined with the learning algorithms. Specifically, we created learning data with four features (the number of appearances, the probability of appearance, the number of transitions, and the transition probability) and compared three machine learning algorithms (RF, SVM, and logistic regression). The accuracy of malware detection by RF, which used the number of system call transitions as a feature, proved to be the highest. Therefore, we utilize this combination in this paper.

Besides, the system calls targeted for feature generation are all system calls that appear in the system call log for both malware and cleanware samples used. For brevity, a total of 176 kinds of system calls have been observed, and the number of transition patterns among two system calls are 30,976 (= 176 × 176).

3.6 Performance Measures

Cross-validation is an efficient way to evaluate the performance of machine learning models. In k-fold cross-validation, a special case of cross-validation, data is divided into k parts; (k − 1) data are used as training data, and the remaining one is used for verification. The model performance is measured k times, and the final performance is given by the average value of k times. In this paper, we consider the 10-fold cross-validation and evaluate the detection results by using a confusion matrix as shown in Table 2. The confusion matrix represents the actual class in the vertical axis and the class predicted by the model in the horizontal axis. In this table, two classes mean malware and cleanware, and each element is described as follows:

**TP: True Positive**

The number of data that are actually malware and correctly classified as malware.

**TN: True Negative**

The number of data that are actually cleanware and correctly classified as cleanware.

**FP: False Positive**

The number of data that are actually cleanware and misclassified as malware.

**FN: False Negative**

The number of data that are actually malware and misclassified as cleanware.

Accuracy, Precision, Recall, and F-measure are calculated and used as classification accuracy indices on the basis of the confusion matrix. More precisely, Accuracy combines the results of the confusion matrix into one numerical value, and indicates the accuracy of malware detection. Precision and Recall are the most commonly used indicators for summarizing the confusion matrix results. The larger the Precision value, the smaller the false positives, and the larger the Recall value, the smaller the detection omissions. Precision and Recall are in a trade-off relationship, and the value indicating the balance between them is F-measure. F-measure is the harmonic mean value of Precision and Recall, and shows how well each datum in the used dataset is classified. The formulas for calculating each value of the above four accuracy indices are shown below:

\[
\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}
\]

(1)

\[
\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}
\]

(2)

\[
\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}
\]

(3)

\[
\text{F-measure} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}
\]

(4)

4. Comparative Experiments

This section describes malware detection using four types of training datasets and compares their performance.

4.1 Experimental Data and Procedures

This paper utilizes the logs of dynamic analysis by Alkanet, which adopts Windows XP Service Pack 3 as its guest OS, hooks invoked system calls by both malware and cleanware samples in this environment, and then records the number of system calls, their arguments, return values, and so on. We analyzed 106 malware samples and 106 cleanware samples. These malware samples, which are Windows executable files, were collected from several sources, such as VirusTotal and computer software companies. Table 3 demonstrates the types of used malware samples, the majority of which is categorized as Trojan. On the other hand, these cleanware samples, which are also Windows executable files, were downloaded from “Mado-no-Mori”\(^1\). The experimental procedure is described below:

1. Extract four patterns of system call sequences from the analysis log of each sample.
2. Create learning datasets using the number of system call transitions as a feature for each pattern.
3. Train the RF model using each dataset created in Step 2 and conduct malware detection using the trained model.
4. Evaluate and compare the detection accuracies in the four patterns described in Sect. 3.4.

4.2 Detection Results

In this subsection, we show the detection results using four

\(^1\)https://forest.watch.impress.co.jp/
Table 3  Types of malware samples.

| Type       | Description                                                                 | Number |
|------------|-----------------------------------------------------------------------------|--------|
| Adware     | A program that can hide on the user’s device and serve the user advertisements | 3      |
| Virus      | A program that can replicate itself and pass malicious code to other non-malicious programs by modifying them | 4      |
| Spyware    | A program that is installed without user’s knowledge, can collect user’s information on the computer and send it to the host computer | 3      |
| Downloader | A program that can allow a user to access a website prepared in advance and download malware | 6      |
| Trojan     | A program that can pretend to be harmless and perform secretly the attacker’s intended behavior | 67     |
| Backdoor   | A program that can bypass a system’s security control, allowing an attacker to access the system stealthily | 4      |
| Ransomware | A program that can infect a user’s computer and display messages demanding a fee to be paid for the system to work again | 2      |
| Worm       | A program that can reproduce itself and spread from computer to computer | 7      |
| Unknown    | Unknown type (family) | 10     |

Table 4  Detection results (Pattern 1).

| Real \ Prediction | Malware | Cleanware |
|-------------------|---------|-----------|
| Malware           | 102     | 4         |
| Cleanware         | 6       | 100       |

Table 5  Pattern 1: Accuracy indices (Pattern 1).

|                  | Precision | Recall | F-measure |
|------------------|-----------|--------|-----------|
| Malware           | 0.94      | 0.96   | 0.95      |
| Cleanware         | 0.96      | 0.94   | 0.95      |

4.2.1 Pattern 1

Table 4 gives the detection results in the case of Pattern 1, in which only the system calls in the primary threads were extracted. From the table, it can be seen that the detection accuracy (i.e., Accuracy) is 95%, and obviously, more malware samples were detected correctly than cleanware samples. The values of other accuracy indices are listed in Table 5. From Table 5, the Recall is higher for malware, and the Precision is higher for cleanware. This implies that the number of samples that are falsely detected as malware is larger than that of samples that are falsely detected as cleanware, but there are few detection omissions in malware cases. The F-measures are the same for both malware and cleanware, which means that each datum in the dataset used can be correctly classified into either malware or cleanware at a rate of 95%.

4.2.2 Pattern 2

Tables 6 and 7 demonstrate the detection results and accuracy indices, respectively, in the case of Pattern 2, where the system calls appearing in all threads were taken into account. In these tables, the detection accuracies considering both malware and cleanware are 94%, and any of the other indices (i.e., Precision, Recall, and F-measure) for both malware and cleanware are the same.

Comparisons between Tables 4 and 6 clearly demonstrate that although Pattern 1 detects malware more accurately than Pattern 2 does, the number of samples that could be detected correctly for both malware and cleanware are the same.

4.2.3 Pattern 3

The detection results stemming from Pattern 3, in which all system calls were extracted from non-primary threads and their detailed accuracy indices, are listed in Tables 8 and 9. From Table 8, we see that the detection accuracy of malware is 90%. Specifically, the number of correctly detected cleanware samples is larger than that of correctly detected malware samples; that is, the false-positive rate in malware detection is higher than that in cleanware detection. It should also be noted that the false-positive rate in malware detection under Pattern 3 is much higher than that under Patterns 1 and 2.

In Table 9, we can see that Precision is higher for malware and Recall is higher for cleanware. These results indicate that a larger number of samples are falsely detected as cleanware compared with the number of samples that are
falsely detected as malware, and the detection omissions in the cleanware case are smaller. Furthermore, since F-measure has a higher value for cleanware than for malware, we conclude that the used dataset can more effectively detect cleanware.

### 4.2.4 Pattern 4

As mentioned in Sect. 3.4, Patterns 1, 2, and 3 consider the system call sequences for samples so that we can detect whether a sample is a malware or a cleanware. However, under Pattern 4, the system call sequences were extracted from every single thread during sample execution. In other words, one can detect whether a thread rather than a sample is malicious or benign. The generated threads for malware and cleanware are 1,301 and 441, respectively. The detection results are listed in Table 10, where “Threads (Malware)” and “Threads (Cleanware)” represent the threads generated during malware execution and cleanware execution, respectively. The Accuracy of this detection is 94%, and the values of other accuracy indices are given in Table 11. As seen from Table 10, 99% of the threads generated when executing the malware samples and 78% of the threads generated when executing the cleanware sample were detected correctly. Table 11 shows that the Recall index is higher for malware threads, and the Precision index is higher for cleanware threads.

Table 10 lists the detection results for the threads, not the samples like other patterns, which makes it difficult to compare the performance of malware detection using the dataset created in Pattern 4 with other patterns. To cope with this issue, subsequent attempts were made to assess whether a sample is a malware or not based on thread detection results; majority voting and threshold-based judgment. In fact, the number of threads generated when the sample is executed would be either odd or even. For example, if two threads were generated during a sample execution and their detection results are [Malicious, Malicious] or [Benign, Benign], the sample can be classified as malware or cleanware soon, whereas if they are [Malicious, Benign] or [Benign, Malicious], such a consensus cannot be reached. Thus, this paper focuses on a threshold-based method, which identifies a sample by comparing the ratio of its threads detected as malicious with a pre-defined threshold value. That is, if the ratio is larger than or equal to the threshold, the sample is identified as malware; otherwise, it is identified as cleanware. The problem now is to find an optimal value of the threshold to obtain the best detection performance considering all accuracy indices. To find such an optimal value, we conducted an experiment varying the threshold value from 0.1 to 1.0. The numerical results are listed in Tables 12 and 13. Note that Table 13 summarizes the accuracy indices in terms of malware.

Although the main objective is to detect the malware, this does not mean that it is acceptable to identify mistakenly cleanware as malware. A more serious problem is misidentifying malware as cleanware. Therefore, in this paper, the optimal threshold is determined by emphasizing the Recall index, which indicates the rate at which a sample that is actually a malware is correctly detected. As these tables show, when the threshold value is 0.5 or lower, all malware samples were correctly detected, and the values of Recall in all cases are 100%. The number of cleanware samples misdetected as malware (i.e., FP) is also large, resulting in low Accuracy and F-measure indices. When the threshold is 0.8 or higher, any accuracy index is high, but the false-positive rates in malware detection are higher compared with those in other cases. When threshold values are 0.6 and 0.7, the num-

| Table 10 | Detection results of threads (Pattern 4). |
|----------|----------------------------------------|
| Real \ Prediction | Malicious | Benign |
| Threads (Malware) | 1286 | 15 |
| Threads (Cleanware) | 96 | 345 |

| Table 11 | Accuracy indices of thread detection (Pattern 4). |
|----------|---------------------------------|
|            | Precision | Recall | F-measure |
| Threads (Malware) | 0.93 | 0.99 | 0.96 |
| Threads (Cleanware) | 0.96 | 0.78 | 0.86 |

| Table 12 | Detection results at each threshold. |
|----------|-----------------------------------|
| TP       | 106 | 106 | 106 | 106 | 106 | 105 | 105 | 103 | 101 | 93 |
| FN       | 0   | 0   | 0   | 0   | 0   | 1   | 1   | 3   | 5   | 13 |
| FP       | 42  | 39  | 33  | 28  | 27  | 15  | 14  | 12  | 12  | 12 |
| TN       | 64  | 67  | 73  | 78  | 79  | 91  | 92  | 94  | 94  | 94 |

| Table 13 | Accuracy indices at each threshold. |
|----------|-----------------------------------|
| Accuracy | 0.802 | 0.816 | 0.844 | 0.868 | 0.873 | 0.925 | 0.929 | 0.929 | 0.920 | 0.882 |
| Precision | 0.716 | 0.731 | 0.763 | 0.791 | 0.797 | 0.875 | 0.882 | 0.895 | 0.894 | 0.886 |
| Recall    | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 0.991 | 0.991 | 0.972 | 0.953 | 0.877 |
| F-measure | 0.835 | 0.845 | 0.865 | 0.883 | 0.887 | 0.929 | 0.933 | 0.932 | 0.922 | 0.882 |
number of misdetected cleanware samples is relatively small, and the values of Recall are high; that is, almost all malware samples can be correctly detected. In particular, a smaller number of cleanware samples were misdetected when the threshold value was 0.7. Also, for a threshold with 0.7, both detection accuracy and F-measure are highest.

Consequently, the value of 0.7 was selected as the optimal threshold, and using this threshold, the detection results of samples and the accuracy indices for sample detection are shown in Tables 14 and 15. From these tables, we can see that the detection accuracy is 93%. Only one malware sample was misdetected, but unfortunately, 14 cleanware samples were misdetected as malware.

### 4.2.5 Comparison among Patterns

From Tables 4, 6, 8, and 14, we see that Pattern 1, which focuses only on the primary threads, has the best detection accuracy. Pattern 4 detects correctly the largest number of malware samples, followed by Pattern 1. However, Pattern 4 demonstrates the largest number of false positives of cleanware. In addition, the number of samples that could be correctly detected as cleanware is larger in both Patterns 1 and 2. Therefore, it is reasonable to say that Pattern 1, in which both malware and cleanware samples were detected with high accuracy, is the best.

The above results show that the information of primary threads is important for malware characterization. Conversely, the results under Pattern 3 imply that the information excluding the primary threads is not a good choice for malware characterization, because this pattern has the worst detection performance among the four patterns. Moreover, under Pattern 4, the detection accuracy of malware is the highest, but the detection accuracy of cleanware is low, compared with those of other patterns, which reveals that the information of single threads are useful for detecting malware, but not for cleanware.

### 4.2.6 Misdetected Threads in Pattern 4

The dataset in Pattern 4 can identify whether a thread is malicious or benign. An interesting question that arises here is how these misdetected threads are generated during malware and cleanware execution.

We first focus on malware and find that 13 out of 106 malware samples contain falsely detected threads. For these 13 malware samples, the number of generated threads and false-positive threads are summarized in Table 16. All samples, including malware and cleanware, were labeled from 0, whereas the false-positive threads were labeled in the order they were created. In this table, each sample contains one false-positive thread, except for the sample with label 44, whose execution generated 258 threads and 3 of them were falsely detected. The ratios of false-positive threads are low for most samples. Furthermore, the misdetection of the primary thread occurred only in the sample with label 94.

Next, let us focus on cleanware, in which 42 out of 106 cleanware samples including false-positive threads were observed. As an example, a small part of the results is given in Table 17. Obviously, the number of threads falsely detected in this case is much larger than that in the case of malware. Moreover, some cleanware samples were identified as malware since all of their generated threads were falsely detected as malicious, for example, the samples with labels 2, 10, 101, and 102. The exact number of such samples is 12, of which only one thread was created in nine samples and two threads were created in the remaining three samples. Compared with malware cases, we see that the number of threads created during the execution of cleanware samples is smaller and these threads are more likely
to be falsely detected. In addition, 25 of the 42 samples including the falsely detected threads were misdetected as malware since their primary threads were classified as malicious, such as the samples with labels 1, 2, 3, 10, 101, and 102 in Table 17, whereas in the case of malware, the sample falsely detected because of misdetecting the primary thread was only the sample with label 94. This implies that the primary thread generated during the execution of the cleanware sample is highly possible to be falsely detected, and might be similar to the primary thread for malware.

5. Discussions

In Patterns 2 and 3, the system calls that appeared in each thread were arranged in the order of issuance, then connected to each other in the order of thread generation, and finally treated as learning data. However, the question of whether the transition between threads can also be regarded as a learning target remains. For example, when thread 2 is created followed by thread 1 (i.e., thread 1 → thread 2) during a sample execution, the transition from the system call invoked at the end of thread 1 to the system call invoked at the beginning of the thread 2 is regarded as a transition pattern for learning, which probably does not exist in the dynamics of the sample. It is not clear whether such a transition affects the detection performance. Therefore, it is necessary to clarify how to handle these thread transitions in the future.

Under Pattern 4, a threshold-based method was adopted for identifying whether a sample is malware or cleanware by using the thread detection results, and an optimal value of threshold was determined by the comparison of detection results in the cases of different thresholds. Nevertheless, as seen in Table 15, although the determined optimal threshold helped detect the malware samples well, it also brought significant false positives in cleanware detection. Therefore, determining the threshold for efficient detection of both malware and cleanware still remains unclear.

Moreover, from Tables 10 and 14, most of the threads generated during the execution of malware samples were correctly detected as malicious, thereby resulting in high malware detection accuracy. On the other hand, the threads created during cleanware execution were largely misdetected as malicious, and the detection accuracy of cleanware under Pattern 4 was much lower than that of other patterns. We consider three possible reasons accounting for the increased number of false positives in the threads created when the cleanware samples were executed: (i) The system call transitions appearing in the cleanware threads were similar to those appearing in the malware thread. If the series of transitions of system calls that appear in the thread are similar, the possibility of false detection is high. (ii) Some transitions of the same system calls appeared in the threads generated when the samples were executed for both malware and cleanware. (iii) The number of threads generated from malware execution was three times larger than that in the case of cleanware. In other words, the bias in learning data between malware and cleanware was large and the learning might be insufficient for cleanware.

In addition, the number of features used this time was 30,976, covering all possible transition patterns between two system calls and resulting in a large time effort for creating four kinds of training datasets and acquiring these detection results. Meanwhile, the features might contain noise data that may have decreased the detection accuracy. That means, to investigate which features are important, i.e., importance analysis of features, and then to omit the unnecessary features become a significant issue and act as one of our future directions.

Finally, it should be noted that the numbers of both malware and cleanware samples used in the experiments were small, which might reduce the reliability of the experimental results. To address this issue, we are continuing to collect data and expect to validate the presented approach with a large-size dataset. Besides, in this paper, only one data set (106 malware samples and 106 cleanware samples) and one learning method (RF) were concentrated. However, it is intuitively that the tendency of the model accuracy might be different with different learning methods and malware samples. Therefore, taking account into the dataset creation for different combinations between different malware samples and learning methods becomes one of the significant topics on malware detection and classification.

6. Conclusions

In this paper, we provided new insight into creating training datasets for efficient malware detection using thread information. Concretely, the thread information was obtained from the dynamic analysis log using Alkanet, and its processing was classified into four patterns, based on whether these threads are primary ones or not. Then, four types of training datasets were created according to these patterns and used for malware detection. In particular, the feature considered was the number of transitions of system calls appearing in the threads. Our comparative experimental results showed that the information of the primary thread was important and useful for malware detection. The method based on the detection results of each thread with a suitable threshold also worked well for malware detection but brought significant false positives in the cleanware detection.

In the future, we would like to clarify these issues stated in Sect. 5 and also compare the detection accuracy in cases of dynamic analysis systems with/without a network connection, aiming to investigate the effect of network connection. Also, this paper focused on only the binary classification of whether a sample is malicious or clean, and although the experimental results showed a high classification accuracy even for the small-size types of malware samples, the class imbalance problem [27] in training dataset highly affects the classification performance so that it should not be ignored. Therefore, when further considering the malware family classification in the future, we would like to collect more malware samples with a relatively good bal-
ance among different malware families (types). Besides, to cope with the malware which has advanced to 64-bit, the updated Alkanet on Windows 10 x64 [28] will be considered.

References

[1] O. Grygorenko, G. Sozonzik, and A.M.K. Al-Dulaimi, “Security problem on the Internet of things networks,” Information and Telecommunication Sciences, vol.2, pp.34–39, 2019.

[2] S. Kothari, G.R. Santhanam, P. Awadhutkar, B. Holland, J. Mathews, and A. Tamrawi, “Catastrophic cyber-physical malware,” Versatile Cybersecurity, Springer, Cham, pp.201–255, 2018.

[3] O. Suciu, S.E. Coull, and J. Johns, “Exploring adversarial examples in malware detection,” Proc. 2019 IEEE Security and Privacy Workshops (SPW’19), IEEE CPS, pp.8–14, 2019.

[4] J. Jung, H.J. Kim, S. Cho, S. Han, and K. Suh, “Efficient Android malware detection using API rank and machine learning,” J. Internet Serv. Inf. Secur., vol.9, no.1, pp.48–59, 2020.

[5] E. Alpaydin, Introduction to Machine Learning, MIT Press, 2020.

[6] Y. Takata, M. Terada, T. Matsuki, T. Kasama, S. Araki, and M. Hatada, “Datasets for anti-malware research–MWS datasets 2018–,” IPSJ SIG Technical Report, vol.2018-CSEC-82, no.38, pp.1–8, 2018. (in Japanese)

[7] M. Ijaz, M.H. Durad, and M. Ismail, “Static and dynamic malware analysis using machine learning,” Proc. 2019 16th International Bhurban Conference on Applied Sciences and Technology (IBCAST’19), IEEE CPS, pp.687–691, 2019.

[8] K. Morimoto, J. Zheng, E. Takimoto, S. Saito, and K. Mouri, “Understanding malware’s actual behaviors from dynamic system call tracer’s log.” International Workshop on Security 2019 (IWSEC’19), Poster Session, no.18, 2019.

[9] Y. Otsuki, E. Takimoto, S. Saito, and K. Mouri, “System call tracer based on virtual machine monitor for malware analysis,” Information Processing Society of Japan, vol.55, no.9, pp.2034–2046, 2014. (in Japanese)

[10] S. Naval, V. Laxmi, M. Rajarajan, M.S. Gaur, and M. Conti, “Employing program semantics for malware detection,” IEEE Trans. Inf. Forensics Security, vol.10, no.12, pp.2591–2604, Dec. 2015.

[11] M. Chowdhury, A. Rahman, and R. Islam, “Protecting data from malware threats using machine learning technique,” Proc. 2017 12th IEEE Conf. Industrial Electronics and Applications (ICIEA’17), IEEE CPS, pp.687–691, 2019.

[12] Y. Kajiwara, J. Zheng, and K. Mouri, “Comparing machine learning algorithms in malware detection based on system calls,” Proc. Forum on Information Technology 2019, vol.L-033, pp.195–198, 2019. (in Japanese)

[13] V. Garg and R.K. Yadav, “Malware detection based on API calls frequency,” Proc. 2019 4th Int. Conf. Information Systems and Comput. Netw. (ISCON’19), IEEE CPS, pp.400–404, 2019.

[14] C. Eagle, The IDA Pro Book, no starch press, 2011.

[15] A.S.M. Ahsan-Ul-Haque, M.S. Hossain, and M. Atiquzzaman, “Sequencing system calls for effective malware detection in Android,” Proc. 2018 IEEE Global Communications Conference (GLOBECOM’18), IEEE CPS, pp.1–7, 2018.

[16] X. Xiao, Z. Wang, Q. Li, S. Xia, and Y. Jiang, “Back-propagation neural network on Markov chains from system call sequences: a new approach for detecting Android malware with system call sequences,” IET Inf. Secur., vol.11, no.1, pp.8–15, 2017. DOI:10.1049/iet-ifs.2015.0211

[17] S.L.S. Darshan, M.A.A. Kumara, and C.D. Jaidhar, “Windows malware detection based on Cuckoo Sandbox generated report using machine learning algorithm,” Proc. 2016 11th Int. Conf. Industrial and Information Systems (ICIS’16), IEEE CPS, pp.534–539, 2016.

[18] M. Wozniak, J. Silka, M. Wieczorek, and M. Alrashoud, “Recurrent neural network model for IoT and networking malware threat detection,” IEEE Trans. Ind. Informat., vol.14, no.8, pp.1–11, 2015.

[19] A. Pektas and T. Acarman, “Malware classification based on API calls and behaviour analysis,” IET Information Security, vol.12, no.2, pp.107–117, March 2018. DOI:10.1049/iet-ifs.2017.0430

[20] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo, and K. Kato, “BitVisor: A thin hypervisor for enforcing IO device security,” Proc. 2009 ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environments, pp.121–130, March 2009.

[21] I. Santos, F. Brezo, J. Nieves, Y.K. Penya, B. Sanz, C. Laorden, and P.G. Bringas, “Idea: Opcode-sequence-based malware detection,” Proc. Int. Symp. Engineering Secure Software and Systems, Springer, Berlin, Heidelberg, pp.35–43, 2010.

[22] D. Canali, A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda, “A quantitative study of accuracy in system call-based malware detection,” Proc. 2012 Int. Symp. Software Testing and Analysis, pp.122–132, July 2012.

[23] M.K. Shankarapani, S. Ramamoorthy, R.S. Movva, and S. Mukkamala, “Malware detection using assembly and API call sequences,” Journal in Computer Virology, vol.7, no.2, pp.107–119, 2011.

[24] S. Tobiyama, Y. Yamaguchi, H. Shimada, T. Ikuse, and T. Yagi, “Malware detection with deep neural network using process behavior,” Proc. 2016 IEEE 40th Annual Computer Software and Applications Conference, pp.577–582, 2016.

[25] X. Jiang, B. Mao, J. Guan, and X. Huang, “Android malware detection using fine-grained features,” Scientific Programming, vol.2020, 2020.

[26] Y. Liu, Y. Wang, and J. Zhang, “New machine learning algorithm: Random forest,” Proc. Int. Conf. Information Computing and Applications, pp.246–252, 2012.

[27] A. Ali, S.M. Shamsuddin, and A.L. Raiescu, “Classification with class imbalance problem: a review,” Int. J. Advances in Soft Computing and Its Applications, vol.5, no.3, pp.1–30, 2013.

[28] Y. Yamashita, J. Zheng, S. Saito, E. Takimoto, and K. Mouri, “Implementation of virtual machine monitor-based stack trace mechanism on Windows 10 x64,” Proc. International MultiConference of Engineers and Computer Scientists 2019, pp.100–105, 2019.

Yuki Kajiwara received the B.S.E. and M.S. degrees in engineering from Ritsumeikan University, Kusatsu, Japan, in 2019 and 2021, respectively. Her research interests include machine learning and computer security especially malware detection. She is a member of Information Processing Society of Japan. She is currently an employee at NEC Corporation.
Junjun Zheng received the B.S.E. degree in engineering from Fujian Normal University, Fuzhou, China, in 2010, and the M.S. and D.Eng. degrees in engineering from Hiroshima University, Higashihiroshima, Japan, in 2013 and 2016, respectively. In 2016 and 2017, he was a Visiting Researcher with the Department of Information Engineering, Graduate School of Engineering, Hiroshima University. Since 2018, he has been an Assistant Professor with the Department of Information Science and Engineering, Ritsumeikan University, Japan. His research interests include performance evaluation and dependable computing. He is a member of the Operations Research Society of Japan; the Reliability Engineering Association of Japan, the Institute of Electrical, Information and Communication Engineers, and the Institute of Electrical and Electronics Engineers.

Koichi Mouri received the B.E., M.E., and Ph.D. degrees in Computer Science from Ritsumeikan University in 1994, 1996, and 2000, respectively. He is a Professor at the College of Information Science and Engineering, Ritsumeikan University, Japan. His research interests include operating systems, computer security and computer network. He is a member of the ACM, IEEE Computer Society, and Information Processing Society of Japan.