Journal of Information Processing Vol.26 461–476 (June 2018)

[DOI: 10.2197/ipsjjip.26.461]

Recommended Paper

Investigation of the Diverse Sleep Behavior of Malware

Yoshihiro Oyama[1,a)]

Received: August 20, 2017, Accepted: March 6, 2018

Abstract: Once malware has infected a system, it may lie dormant (or asleep) to control resource consumption speeds, remain undetected until the time of an attack, and thwart dynamic analysis. Because of their aggressive and abnormal use of sleep behavior, malware programs are expected to exhibit traits that distinguish them from other programs. However, the details of the sleep behavior of real malware are not sufficiently understood, and the diversity of sleep behavior among different malware samples or families is also unclear. In this paper, we discuss the characteristic sleep behavior of recent malware and explore the potential for applying the features of sleep behavior to malware classification. Specifically, we demonstrate that a wide variety of sleeps are executed by a set of malware samples and that sleeps are a promising source of features for distinguishing between different malware samples. Furthermore, we show that applying a learning algorithm to sleep behavior information can result in high classification accuracy and present several examples of typical and rare sleep behaviors observed in the execution of real malware.

Keywords: malware, sleeps, dynamic analysis, malware classification, anti-analysis

1. Introduction

Malware programs may execute sleeps, both for similar reasons as normal programs (such as waiting for an event), and for reasons that are specific to malware (such as evading security systems). For example, malware can leverage a long sleep to avoid being detected or cause a time-out in a dynamic analysis performed by a security system. In addition, malware can sometimes infer the existence of an analysis system, such as a sandbox or a hypervisor, by measuring the difference in the system time before and after sleep. Various studies have found that modern malware execute anti-analysis operations—operations that are carried out by the malware to evade being analyzed [4], [5], [6], [9], [12], [13], [23], [26], [30], [33], [40]. One of the most frequently observed anti-analysis operations is to go to sleep [4], [6], [26], [33], and many analysis or protection systems are equipped with mechanisms to counteract obstructive sleeps executed by malware [17], [27], [29], [35].

Considering the aggressive and abnormal use of the sleep function by malware, we can expect the sleep behaviors of malware programs to be more characterizable than those of normal programs. However, the sleep behaviors of real malware have not been comprehensively studied and are not fully understood. The diversity of sleep behaviors across different malware samples or families also remains unclear. Consequently, techniques for accurately estimating the intent underlying individual sleep executions is still at its developmental stage. Such a technology can be valuable, for example, to accelerate dynamic analysis by skipping analysis-hindering sleeps and executing other sleeps precisely. However, to the best of our knowledge, no state-of-the-art analysis technology can distinguish between these two types of sleeps with reasonable accuracy.

The accurate detection and classification of sophisticated modern malware has long been a significant challenge, and methods that improve accuracy are constantly required. A deeper understanding of the sleep behavior of malware is likely to aid our comprehension of the sophisticated and complicated anti-analysis operations used by modern malware and can therefore lead to new techniques for malware detection or classification. While the aspects of program behavior examined by traditional detection or classification techniques, such as file I/O and registry key accesses, provide important clues into potentially malicious programs, the understanding of the sleep behaviors of such programs can be significant for the exploration of new clues, and to integrate the resulting findings into existing techniques.

In this paper, we describe the characteristic sleep behavior of recently collected malware and classify malware samples based on these characteristics. The goal of this study is to highlight the diversity of sleep behavior and its potential for malware classification. To this end, we first extracted information on API calls related to sleeps from the logs of the FFRI Dataset 2016 [21], which is a dataset of dynamic analysis logs of Windows malware. We then defined a set of features that are expected to prove effective for classification of malware and calculated the feature values of each malware sample using the extracted information. Example features include the maximum amount of time to sleep, number of calls to a sleep function, earliest call positions of a sleep function, and types of API functions called just before or after sleeps. This paper summarizes the feature values, which we believe uncover the quantitative trends of sleeps executed by recent malware. We hypothesize that the diversity among samples within the same

[1] University of Tsukuba, Tsukuba, Ibaraki 305–8577, Japan.
[a)] oyama@cc.tsukuba.ac.jp

The preliminary version of this paper was presented at the 76th meeting of SIGCSEC in March 2017, and recommended to be submitted to Journal of Information Processing (JIP) by the chief examiner of SIGCSEC.

malware family is smaller than that among samples from different malware families. Thus, the similarity of sleep behaviors can be leveraged effectively for malware classification. We supplied four widely used learning algorithms (support vector machine (SVM), random forest (RF), naive bayes (NB), and C5.0) with the feature values and malware family names and set up the algorithms to create models for classifying malware samples using only the feature values based on sleep behavior. Then, we measured the precision and recall of the classification achieved by the models in cross-validation tests. Our results show that RF was the best of the algorithms in our experiments, achieving 83.3% precision and 74.8% recall using only sleep-related information.

Although many studies have quantitatively investigated the anti-analysis operations executed by malware [5], [6], [9], [12], [13], [33], to the best of our knowledge, no study has yet examined the sleep behaviors of malware programs. Furthermore, no study has attempted to classify malware samples through extensive use of sleep information. Therefore, the novelty of this study is twofold: (i) this is the first time that sleep behavior has been used as a malware classification tool; (ii) promising classification results based on the extensive use of sleep information are presented.

The remainder of this paper is organized as follows. Section 2 briefly describes the FFRI Dataset, and Section 3 introduces background knowledge on sleep behavior concerned with our study. Section 4 summarizes the measured feature values and the classification results given by the learning algorithms. Section 5 discusses some areas that require further attention, and Section 6 describes related studies. Finally, Section 7 summarizes this paper and briefly presents directions for future work.

2. The FFRI Dataset

The FFRI Dataset [21] contains dynamic malware analysis logs obtained by FFRI, Inc., using Cuckoo Sandbox [17]. In this study, we use the latest version (FFRI Dataset 2016) and choose logs collected on Windows 10 (x64). The dataset contains the analysis results of 8,243 malware samples collected from January–March 2016. All were judged as malware by more than 10 anti-malware products. Each sample was executed for up to 90 s, and any malware process was terminated after 90 s had elapsed. The execution environment provided a network connection for the samples, and some of them successfully communicated with external hosts. **Table 1** presents some basic information of the dataset used in this study. The dataset contains API call sequences of benign processes of `lbass.exe`, and we first excluded such sequences from the dataset.

Table 1 Basic information on FFRI Dataset 2016 Windows 10 (x64).

Number of malware samples	8,243
Number of API functions used by the entire set of malware samples	288
Number of API functions used by one malware sample	Maximum: 140 Average: 47.5 Minimum: 0
Number of API calls invoked by one malware sample	Maximum: 83,444 Average: 4,569 Minimum: 0

3. Sleeps

3.1 API Functions for Sleeps

Windows operating systems provide multiple API functions for sleeps. Among them, `NtDelayExecution` is the only one recorded in FFRI Dataset 2016. `Sleep` and `SleepEx` are higher-level API functions for sleeps, and are not recorded. These functions invoke `NtDelayExecution`, the arguments of which are recorded by Cuckoo Sandbox. One of the arguments is the amount of time to sleep (or simply *sleep time*) in milliseconds (the resolution). A sleep time of zero can be set, and zero millisecond sleeps are mainly used for voluntary thread context switches.

This study does not deal with sleeps that are virtually achieved with another method. For example, if malware repeatedly calls a time measurement function `GetSystemTimeAsFileTime` until a predetermined amount of time has elapsed, it can sleep for that amount of time. Calls to an I/O-related API function with a timeout argument can also be used as a virtual sleep. Time-consuming computations or a combination of timers and wait operations can also be substituted for sleep, depending on the circumstances and demands [43]. It is not straightforward to determine whether a given API call sequence is for sleeps or for another purpose. Considering such sleeps significantly complicates the analysis of sleep behavior and obscures the findings. Understanding obvious sleeps is sufficiently challenging, and this study focuses on these clearer instances as the first step toward long-term research goals.

3.2 Purposes of Sleep

The major purposes of sleep operations for malware are listed as follows. The first and second purposes are not specific to malware in that they are also relevant for general software, whereas the others are specific to malware.

Control of resource consumption speeds Programs can sleep to yield resources to other processes or threads. These sleeps suppress an increase in CPU load or network load, and support malware in avoiding detection and preventing a denial of service because of a high load. In some call sequences in the dataset, multiple name resolution requests to DNS servers are executed at 1-s intervals. In other sequences, multiple `NtGetContextThread` calls are repeatedly invoked to obtain the status of another thread at intervals of 10 ms. The malware may wish to obtain the status to determine whether the execution of the thread is desirable. In other sequences, multiple `GetFileAttributesW` calls to a non-existent file are repeatedly invoked at 5-s intervals. The malware may be synchronizing with other processes or threads based on the existence of the file.

Execution control and synchronization of threads A thread can control the execution of other threads or synchronize with other threads by suspending and resuming the threads at certain intervals. In some call sequences, a main thread repeatedly executes a sequence of invoking `NtSuspendThread`, `NtDelayExecution`, and `NtResumeThread` for another thread. The combination of `NtSuspendThread` and `NtResumeThread` forces the thread to alternate between sleep and execution, and the times

Table 2 Features of sleep behavior.

ID	Name	Definition
(1)	maxtime	Maximum sleep time provided to `NtDelayExecution`
(2)	mintime	Minimum sleep time provided to `NtDelayExecution`
(3)	avetime	Average sleep time provided to `NtDelayExecution`
(4)	modetime	Mode of sleep times provided to `NtDelayExecution`
(5)	ntimekinds	Number of distinct sleep times provided to `NtDelayExecution`
(6)	maxchunktime	Maximum of compound sleep times provided in a chunk
(7)	minchunktime	Minimum of compound sleep times provided in a chunk
(8)	avechunktime	Average of compound sleep times provided in a chunk
(9)	modechunktime	Mode of compound sleep times provided in a chunk
(10)	nchunktimekinds	Number of distinct compound sleep times provided in a chunk
(11)	ncalls	Number of `NtDelayExecution` calls
(12)	nprocs	Number of processes that call `NtDelayExecution`
(13)	nthreads	Number of threads that call `NtDelayExecution`
(14)	nchunks	Number of chunks
(15)	maxchunksize	Maximum chunk size
(16)	unroundedratio	Ratio of `NtDelayExecution` calls to which unrounded (nonzero last digit) sleep times are provided
(17)	earliestsleep	Earliest call position of `NtDelayExecution` in a call sequence among all threads
(18)	maxsleepratio	Maximum ratio of `NtDelayExecution` calls in a call sequence among all threads
(19)	ninterrupts	Number of `NtDelayExecution` calls interrupted by asynchronous procedure calls
(20)	nsleepsatend	Number of threads that execute `NtDelayExecution` at the end of the call sequence
(21)	progression	Whether a set of sleep times include an arithmetic sequence composed of at least ten numbers
(22)	communication	Whether a communication-related API (e.g., `gethostbyname`) is called just before or after a chunk
(23)	window	Whether a window-related API (e.g., `GetForegroundWindow`) is called just before or after a chunk
(24)	thread	Whether a thread-related API (e.g., `NtSuspendThread`) is called just before or after a chunk
(25)	fileattr	Whether a fileattr-related API (e.g., `GetFileAttributesW`) is called just before or after a chunk

provided to the `NtDelayExecution` calls work as the "time slices" provided to the thread. In addition, execution of extremely long sleeps can also be used to effectively wait for an event.

Delitescence Malware can hide itself with a long sleep until some condition triggers the start of its activity. Some malware avoid early detection by simply executing a long sleep, and others execute a sleep in a polling loop, waiting for a trigger to start an activity. The time bomb is a well-known instance of malware delitescence. A recent example of malware that executes a long sleep is the KeRanger ransomware, which first sleeps for three days and then executes its encryption routine [42].

Time-out of dynamic analysis Malware can execute a long sleep to cause dynamic analysis to time-out. A time limit, typically from a few minutes to several tens of minutes, is often imposed on the duration of the analysis of the malware. Malware can evade the analysis of its critical program parts by sleeping for an amount of time exceeding the time limit. A recent example is the Neutrino POS-terminal trojan, which executes a random-time sleep before it starts its malicious routine, possibly to evade sandboxes [50]. Some call sequences in the dataset seem to be directed toward delitescence or a time-out. In addition, the dataset contains a record of sleeps that were skipped by the anti-sleep mechanism of Cuckoo Sandbox. This mechanism skips some of the sleeps attempted within the first 5 s of a process execution.

The purposes and typical operations of delitescence and time-out of dynamic analysis are similar. Hence, they can be identified identically. However, they can also be regarded as distinct from the viewpoint of the expected durations of sleep and the finishing triggers for the sleeps. The durations of sleep required for delitescence are usually in the order of days, whereas those for time-outs are in the order of minutes. Sleeps for delitescence can be ended with various triggers including network communication, whereas triggers for time-out are simpler, such as the passage of a certain amount of time.

Detection of analysis systems Malware can detect time-distorting operations executed by a dynamic analysis system, such as sleep skips, by comparing the specified sleep time and the actual time elapsed during the sleep [26].

4. Measurement

4.1 Calculation of Feature Values

4.1.1 Methods

Out of 8,243 malware samples, we selected 1,234 target samples whose execution involves a process-wise cumulative sleep time greater than or equal to 60 s. Cumulative sleep time is the sum of sleep times provided to an argument of `NtDelayExecution`. We consider that these samples are likely to make more effective use of sleep than others.

Table 2 presents the features defined in this study. These features were selected because, through an elaborate reading of call sequences close to `NtDelayExecution`, we conjectured that different malware families are likely to exhibit diverse values for these features.

We now explain several of these features. We refer to consecutive `NtDelayExecution` calls in each thread's API calls as a chunk of sleeps, or simply, a *chunk*. The *size* of a chunk is the number of `NtDelayExecution` calls contained in the chunk. An isolated `NtDelayExecution` call is regarded as a chunk of size one. **Figure 1** shows an example of call sequences in the dataset that contain chunks. There are four chunks in this example: sizes

Table 3 Feature values through all target malware samples.

	Max	Average	Min
Maximum sleep time in each sample (ms)	2,728,163,227	6,654,420.6	5
Maximum sleep time in each sample (except extreme samples) (ms)	65,000	21,986.5	5
Minimum sleep time in each sample (ms)	240,000	13,738.4	0
Minimum sleep time in each sample (except extreme samples) (ms)	65,000	13,381.9	0
Average sleep time in each sample (ms)	1,364,201,613	2,396,366.5	5
Average sleep time in each sample (except extreme samples) (ms)	65,000	15,316.1	5
Average compound sleep time in each chunk in each sample (ms)	2,728,403,227	4,995,263.0	15
Average compound sleep time in each chunk in each sample (except extreme samples) (ms)	88,347	20,557.9	15
Maximum chunk size in each sample	15,118	504.8	1
Number of distinct sleep times in each sample	138	9.0	1
Number of `NtDelayExecution` calls in each sample	15,118	600.8	1
Number of chunks in each sample	5,184	60.2	1
Average of all sleep times in all samples (ms)	11,229.1		
Average of all sleep times in all samples (except extreme samples) (ms)	189.8		
Total number of distinct sleep times in all samples	2,636		
Total number of `NtDelayExecution` calls in all samples	741,440		
Total number of chunks in all samples	74,295		

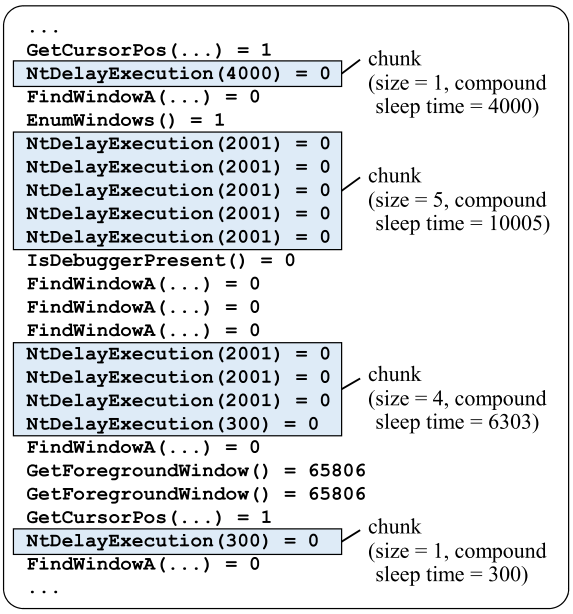


Fig. 1 API call sequence containing chunks.

1, 5, 4, and 1. Large chunks appear for various reasons. For example, some malware use a sequence of short sleeps in order to effectively execute a long sleep, because some analysis systems skip the executions of long sleeps. We introduce the "unroundedratio" feature because, in the dataset, a large number of samples adhere to unrounded sleep times only and many other samples adhere to rounded sleep times only. We introduce the "earliestsleep," "maxsleepratio," and "nsleepsatend" features because we expect these to characterize specific malware purposes such as delaying analysis, hiding until triggered, and attempting timing-based sandbox detection. We introduce the "ninterrupts" feature because interrupted sleeps can be related to some specific malware operations. The "progression" feature is used because some malware samples invoke `NtDelayExecution` with successively increasing sleep times, such as 0, 3, 6, . . . , 42, and 45 ms. We introduce the "communication," "window," "thread," and "fileattr" features because they provide essential hints regarding the purpose of the sleeps.

Table 4 Top 10 sleep times in terms of frequency of occurrences.

Sleep time	# of occurrences	# of samples	# of families
50 ms	410,032	395	13
5 ms	149,714	32	7
10 ms	28,128	421	18
1 ms	22,220	46	8
25 ms	19,787	15	4
1,000 ms	17,548	566	28
100 ms	11,835	64	13
12 ms	10,233	64	7
5,000 ms	7,588	593	28
0 ms	6,547	128	24

4.1.2 Results

Table 3 summarizes the feature values calculated from the logs of all target samples. The columns Max, Average, and Min indicate the maximum, average, and minimum values, respectively, calculated using the feature values of malware samples. We calculated two versions of values for several features. The first version of the values was computed without any adjustment, whereas for the second version of the values, we eliminated the call sequences of three extreme samples that contain `NtDelayExecution` calls with sleep times of 2,728,163,227 ms (approximately 31 days), because these are much larger than the other times.

The results show that the sleep times range widely from 0 to 2,728,163,227 ms, with numerous distinct sleep times used (2,636 distinct values). The average of all sleep times in all samples except the extreme samples is 190.4 ms. Although the target samples were selected because of their long cumulative sleep times, long sleeps such as a few seconds are not dominant in the samples. However, there remains a possibility that many samples achieve substantively long sleeps by forming chunks of large sizes. One sample uses 138 distinct sleep times. The maximum chunk size is 15,118, with one sample continuously invoking a significantly large number of `NtDelayExecution` calls.

We investigated the distribution of sleep times. **Table 4** lists the top 10 sleep times (in terms of frequency of occurrence) observed in the target sample logs. The number of 50 ms sleeps is by far the

largest, although 5,000 ms sleeps are used by the largest number of malware samples. Sleeps of 1,000 ms and 5,000 ms are used by the largest number of malware families, whereas 5 ms, 1 ms, and 25 ms sleeps are frequently used by a relatively smaller number of malware samples or families. The use of 0 ms is not rare and can be observed in many malware families. Overall, rounded and small numbers such as 1, 5, 10, and 50 occur frequently.

4.2 Classification of Samples

4.2.1 Methods

We classified malware samples based on the feature values outlined above. The training data are vectors of feature values and their labeling results, which associate each vector with the malware name determined by an anti-malware product. Although the dataset contains the labeling results of many anti-virus products, we chose a Microsoft product because of its global dominance. We refer to the classification of samples according to Microsoft's anti-malware software, along with the names it uses to identify the malware, as the *Microsoft classification*.

Many subsets of the investigated malware set have malware names from the same family in Microsoft's malware naming scheme [31]. For example, the set contains Trojan:Win32/Matsnu.M and Trojan:Win32/Matsnu.O samples, both of which are from the Trojan:Win32/Matsnu family. In this scheme, the malware names are composed of different name parts, referred to as type, platform, family, variant, and information. In this study, we regard combinations of type, platform, and family as malware (family) names, and consider samples with the same malware name to be variants from the same family.

Among the abovementioned 1,234 samples, 147 were not determined as malware by the Microsoft product. Hence, we exclude these samples and use the remaining 1,087 in the study. The number of samples belonging to each malware family is listed in **Table 5**.

Table 5 Number of samples from each malware family.

Malware name	# of samples
TrojanSpy:Win32/Ursnif	281
Worm:Win32/Gamarue	178
Trojan:Win32/Dynamer	147
TrojanDropper:Win32/Rovnix	74
Trojan:Win32/Skeeyah	70
Ransom:Win32/Crowti	40
Trojan:Win32/Tinba	40
Ransom:Win32/Teerac	30
Ransom:Win32/Isda	26
PWS:Win32/Fareit	25
Trojan:Win32/Bagsu	24
Trojan:Win32/Avkill	14
Trojan:Win32/Matsnu	14
TrojanDownloader:Win32/Dofoil	10
TrojanSpy:Win32/Skeeyah	10
Trojan:Win32/Bulta	8
VirTool:Win32/CeeInject	8
Backdoor:Win32/Fynloski	6
Backdoor:Win32/Qakbot	6
TrojanDownloader:Win32/Banload	6
5 malware families	3 each
13 malware families	2 each
29 malware families	1 each
Sum	1,087

We randomly selected one sample from each malware family. **Table 6** presents the feature values of some of the randomly selected representative samples, and **Table 7** summarizes the feature values of all 1,087 samples used. The top rows in the tables indicate the identification numbers of features and the leftmost column in Table 6 indicates the identification numbers of malware samples. The word "true" in the tables in this paper indicates the satisfaction of the corresponding condition, whereas the single period indicates that the condition was not satisfied. The Average' and Max' rows in Table 7 indicate the values calculated excluding the extreme samples described in Section 4.1.2. The variations in the values in the tables suggest that the sleep behavior of the samples is significantly diverse. Nevertheless, distinguishing between some pairs of samples using only these feature values is a difficult task. For example, all feature values of samples 12 and 13 coincide. The feature values of all 1,087 samples are available at https://www.dropbox.com/s/1fc4w5ni1adp3gt/sleep_feature_values_1087.csv?dl=0.

Next, we created classification models with four learning algorithms: SVM, RF, NB, and C5.0. SVM [8], [16] represents input data as points in space and creates a hyperplane to divide them into two classes, maximizing the distances between the points and the hyperplane. The hyperplane then works as the classifier of unknown data. An advantage of SVM is its significantly high classification ability. RF [10] creates multiple decision trees based on randomly sampled data. When it classifies unknown data, it integrates the outputs from the trees and makes a decision based on the integrated value. An advantage of RF is that it has a mechanism to reduce the impact of overfitting. NB [18], [38] computes a set of conditional probabilities that relate the occurrence probabilities of the input data and that of their classes, and then creates a probabilistic classifier based on the values. Although NB has the advantage of being simple and fast, its classification ability tends to be lower than that of advanced algorithms such as SVM. C5.0 [36], [37] creates a decision tree using the difference between the information entropy computed before and after the introduction of a potential branch. An advantage of C5.0 is that it generates an intuitively understandable output.

The inputs to the algorithms are vectors of feature values extended with malware (family) names. We provided 1,087 feature vectors, each of which has 25 feature values and one malware name. The code for model creation and testing uses the R language, with the `kernlab` package to implement the SVM, `e1071` to implement RF and NB, and `C50` to implement C5.0.

We evaluated the resulting classification models with cross-validation tests in which we provided feature vectors to each algorithm and had them label the samples represented by the vectors. In the tests, we first excluded all malware families that contained fewer than 10 samples, because these sets are too small for cross-validation tests. The number of excluded samples was 104. We then randomly partitioned the remaining 983 samples into four subsets and conducted fourfold cross-validation. In each subtest, three subsets were used as training data and the remaining subset was used as test data. Finally, we had the models label samples in the test data with the names of samples in the training data. The

Table 6 Feature values of samples selected randomly from each malware family.

	Malware name	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)
1	TrojanDownloader:Win32/Banload	61,000	3,000	14,600.0	3,000	2	61,000	3,000	14,600.0	3,000	2
2	TrojanDropper:Win32/Rovnix	60,000	60,000	60,000.0	60,000	1	60,000	60,000	60,000.0	60,000	1
3	VirTool:Win32/CeeInject	60,000	60,000	60,000.0	60,000	1	60,000	60,000	60,000.0	60,000	1
4	Trojan:Win32/Bulta	60,000	60,000	60,000.0	60,000	1	60,000	60,000	60,000.0	60,000	1
5	Worm:Win32/Gamarue	60,000	15,000	37,500.0	60,000	2	60,000	15,000	37,500.0	60,000	2
6	TrojanDownloader:Win32/Dofoil	30,000	100	607.8	100	4	60,000	250	10,687.5	6,000	4
7	Backdoor:Win32/Qakbot	30,000	10	1,191.7	10	4	30,000	10	1,214.6	10	5
8	Trojan:Win32/Matsnu	24,000	4	236.0	12	11	83,000	11	7,809.1	5,000	15
9	Backdoor:Win32/Dodiw	20,000	0	866.5	300	11	20,000	0	866.5	300	11
10	TrojanSpy:Win32/Skeeyah	10,000	10,000	10,000.0	10,000	1	10,000	10,000	10,000.0	10,000	1
11	Trojan:Win32/Tinba	10,000	1	144.2	1	9	60,000	24	1,939.0	24	8
12	Ransom:Win32/Crowti	10,000	0	7,888.9	10,000	3	10,000	0	7,888.9	10,000	3
13	Trojan:Win32/Bagsu	10,000	0	7,888.9	10,000	3	10,000	0	7,888.9	10,000	3
14	PWS:Win32/Fareit	5,000	5,000	5,000.0	5,000	1	5,000	5,000	5,000.0	5,000	1
15	TrojanDownloader:Win32/Zemot	5,000	250	4,736.1	5,000	2	5,000	250	4,736.1	5,000	2
16	Ransom:Win32/Isda	5,000	100	2,869.6	5,000	2	5,000	1,000	4,714.3	5,000	2
17	Trojan:Win32/Dynamer	5,000	10	2,524.2	5,000	6	5,000	10	2,524.2	5,000	6
18	Trojan:Win32/Avkill	5,000	10	1,038.1	1,000	3	76,000	10	20,502.5	76,000	4
19	TrojanSpy:Win32/Ursnif	5,000	10	138.9	50	6	60,100	10	2,665.6	1,000	7
20	Trojan:Win32/Skeeyah	5,000	10	119.1	50	5	66,000	10	4,609.1	5,000	5
21	Ransom:Win32/Teerac	5,000	0	4,458.3	5,000	3	5,000	0	4,458.3	5,000	3
22	Backdoor:Win32/Kasidet	5,000	0	1,647.5	5,000	18	5,000	10	4,248.9	5,000	3
23	Backdoor:Win32/Fynloski	500	10	330.1	200	3	500	10	330.1	200	3
24	Ransom:Win32/Troldesh	100	100	100.0	100	1	78,700	78,700	78,700.0	78,700	1
25	Trojan:Win32/MultiInjector	25	25	25.0	25	1	70,500	70,500	70,500.0	70,500	1

	(11)	(12)	(13)	(14)	(15)	(16)	(17)	(18)	(19)	(20)	(21)	(22)	(23)	(24)	(25)
1	5	1	2	5	1	0%	40	1%	0	0	.	.	.	.	.
2	1	1	1	1	1	0%	153	0%	0	0	.	.	.	.	.
3	1	1	1	1	1	0%	411	0%	0	0	.	.	.	.	.
4	1	1	1	1	1	0%	410	0%	0	0	.	.	.	.	.
5	2	1	1	2	1	0%	133	0%	0	0	.	.	.	.	true
6	211	2	4	12	100	0%	4	40%	1	0	.	.	.	.	.
7	53	3	5	52	2	0%	18	8%	0	0	.	.	.	.	.
8	695	3	5	21	187	27%	2	99%	1	0	.	true	.	true	true
9	307	1	6	307	1	0%	3	46%	0	1	.	true	true	.	true
10	7	1	1	7	1	0%	430	1%	0	0	.	true	.	.	.
11	1,170	4	11	87	60	22%	2	49%	0	0	.	true	.	true	.
12	9	3	3	9	1	0%	163	1%	0	0	.	true	.	.	.
13	9	3	3	9	1	0%	163	1%	0	0	.	true	.	.	.
14	13	1	1	13	1	0%	2,009	1%	0	0	.	true	.	.	.
15	18	1	3	18	1	0%	27	3%	1	0	.	true	.	.	.
16	23	2	2	14	10	0%	10	10%	0	0	.	.	.	.	.
17	36	2	7	36	1	0%	1	48%	2	0	.	true	.	true	true
18	79	1	2	4	76	0%	34	34%	0	0	.	.	.	.	.
19	1,267	2	7	66	1,202	0%	1	100%	1	0	.	true	.	true	true
20	1,354	2	5	35	1,320	0%	1	100%	0	0	.	true	.	true	true
21	18	2	3	18	1	0%	126	3%	1	0	.	true	.	.	.
22	49	2	2	19	16	78%	1	9%	0	0	true	true	true	.	.
23	694	3	4	694	1	0%	1	33%	0	0	.	true	true	.	.
24	787	1	1	1	787	0%	300	72%	0	0	.	.	.	.	.
25	2,820	1	1	1	2,820	100%	60	98%	0	0	.	.	.	.	.

final precision and recall values were calculated by taking the average of the four values obtained in the subtests. If the diversity of sleep behavior among variants in the same malware family is small and the diversity among samples from different families is large, then the models are likely to assign the correct names to most samples.

The classification models are evaluated with micro-averages and macro-averages of the precision and recall values. Here, *precision* is defined as the proportion of samples assigned the correct malware names in each subset. *Recall* is defined as the proportion of samples assigned the correct malware names in each subset created by the Microsoft classification. *Micro-averages* of precision and recall are calculated using the sum of the number of samples in each subset of the classification results, whereas the *macro-averages* of precision and recall are the average values for each subset. For the tests, the micro-average of precision and the micro-average of recall are always the same.

We calculate both micro-averages and macro-averages because

Table 7 Summary of feature values of target malware samples determined as malware by the anti-malware product.

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)
Average	2,532,442.9	14,095.1	209,145.2	17,107.4	8.6	2,558,460.3	15,797.8	648,208.3	2,534,410.4	4.0
Average'	22,653.9	14,108.1	16,097.5	17,123.2	8.6	48,695.4	15,812.3	20,774.5	24,623.3	4.0
Median	10,000	10	1,750.2	1,000	4	60,000	11	5,000.0	5,000	4
Mode	5,000	10	60,000.0	50	1	60,000	10	60,000.0	5,000	1
Max	2,728,163,227	65,000	209,858,879.0	65,000	138	2,728,163,227	88,000	682,041,356.8	2,728,163,227	41
Max'	65,000	65,000	65,000.0	65,000	138	88,694	88,000	88,347.0	88,694	41
Min	5	0	5.0	0	1	57	0	52.4	0	1

	(11)	(12)	(13)	(14)	(15)	(16)	(17)	(18)	(19)	(20)	(21)	(22)	(23)	(24)	(25)
Average	558.2	1.6	3.9	52.0	470.2	11%	201.6	47%	0.2	0.1					
Average'	558.7	1.6	3.9	52.1	470.6	11%	201.8	47%	0.2	0.1					
Median	74	2	3	16	2	0%	7	36%	0	0					
Mode	1	1	1	1	1	0%	1	100%	0	0					
Max	15,015	5	102	1,485	15,015	100%	18,313	100%	2	29					
Max'	15,015	5	102	1,485	15,015	100%	18,313	100%	2	29					
Min	1	1	1	1	1	0%	1	0%	0	0					
True / False											6 / 1,081	572 / 515	58 / 1,029	516 / 571	390 / 697

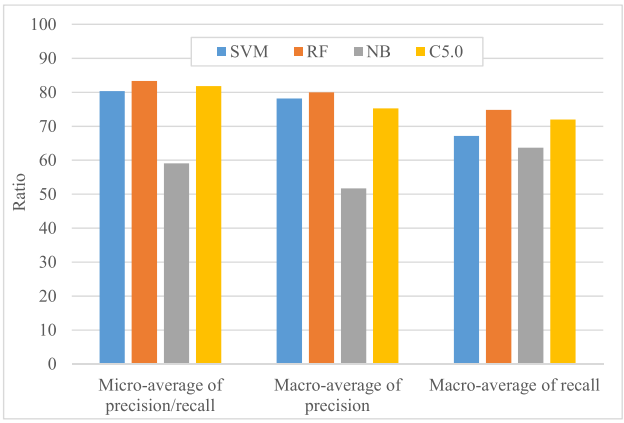


Fig. 2 Results of cross-validation tests.

we intend to clarify the effect on results caused by the difference in the numbers of samples in each malware family. Because the number of samples in each malware family is not uniform, we consider not only the proportion of correct answers obtained through the classification of all samples but also the proportion of correct answers in each of the resulting sample sets and in each of the Microsoft classification sample sets. The macro-averages of precision and recall provide an indication of these proportions.

4.2.2 Results

The cross-validation test results are presented in **Fig. 2**. The micro-averages of precision and recall exceeded 80% when using any of SVM, RF, and C5.0. These results indicate that the sleep behavior of malware samples is sufficiently diverse to distinguish between many of them, and that the sleep behavior of a malware sample is considerably correlated with that of its family. We believe that the precision and recall values resulting from these algorithms indicate the strong potential for using sleep behavior in malware classification. In contrast, NB provided the poorest values; the micro-average and macro-average of precision were less than 60% and the macro-average of recall was approximately 64%. RF was the best algorithm in terms of all measures. When using RF, the micro-average of precision, macro-average of precision, and macro-average of recall were 83.3%, 80.0%, 74.8%, respectively.

Table 8 shows the details of the classification from the four algorithms. The indexes from (a) to (o), both in the top row and in the leftmost column, represent different malware, the names of which are indicated in the leftmost column. The top row indicates the malware based on the Microsoft classification, whereas the leftmost column indicates decisions by the corresponding algorithm. In other words, each column contains the classification decisions of the algorithm for the specific malware represented by that column, and each row contains the breakdown of actual malware samples that the algorithm labels with the specific malware name written in the leftmost cell of the row.

SVM, RF, and C5.0 could correctly label most samples of Ursnif, Gamarue, Rovnix, Crowti, and Tinba. This successful labeling greatly raises the micro-average of precision and recall because the dataset contains many samples of these malware families. By contrast, among such large malware families, these algorithms incorrectly labeled many samples of Dynamer and Skeeyah. They incorrectly identified Dynamer as Ursnif or Skeeyah, and incorrectly identified Skeeyah as Ursnif or Dynamer. We speculate that many samples of Ursnif, Dynamer, and Skeeyah have similar sleep behavior. In addition, NB provided wrong labeling to many of the Gamarue samples; it incorrectly identified 140 of 178 Gamarue samples as Rovnix. This classification error significantly lowers the micro-average of precision and recall of NB. NB also incorrectly classified many samples belonging to small families. All algorithms fared poorly in labeling Bagsu, providing incorrect classifications for most Bagsu samples. It is important for future work to scrutinize the API call sequences of the samples that were incorrectly labeled by the algorithms, and to compare them with the sequence of the malware families that the samples were incorrectly classified as.

Next, we present the results of the tests in which we varied the threshold of cumulative sleep time to select target samples. The purpose of these tests is to identify the effect of the threshold on classification results. **Table 9** shows the number of samples classified in the tests, and **Fig. 3** shows the results. Overall, the effect

Table 8 Classification of each malware family.

(a) SVM

	(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)	(i)	(j)	(k)	(l)	(m)	(n)	(o)	Sum
(a) TrojanSpy:Win32/Ursnif	278	0	31	0	12	0	0	0	0	0	0	0	0	0	0	321
(b) Worm:Win32/Gamarue	0	165	5	4	8	0	0	0	0	0	13	0	0	0	1	196
(c) Trojan:Win32/Dynamer	3	7	80	0	14	1	1	0	1	7	4	1	0	5	6	130
(d) TrojanDropper:Win32/Rovnix	0	1	1	70	5	0	0	0	0	0	0	0	0	0	0	77
(e) Trojan:Win32/Skeeyah	0	1	19	0	22	0	0	0	0	2	0	0	0	0	3	47
(f) Ransom:Win32/Crowti	0	0	1	0	0	39	0	0	0	0	1	0	0	0	0	41
(g) Trojan:Win32/Tinba	0	0	0	0	5	0	39	0	0	0	2	0	0	0	0	46
(h) Ransom:Win32/Teerac	0	0	2	0	1	0	0	30	0	0	1	0	0	2	0	36
(i) Ransom:Win32/Isda	0	3	1	0	2	0	0	0	24	0	2	1	0	2	0	35
(j) PWS:Win32/Fareit	0	0	2	0	0	0	0	0	0	16	0	0	0	0	0	18
(k) Trojan:Win32/Bagsu	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	2
(l) Trojan:Win32/Avkill	0	0	1	0	1	0	0	0	0	0	0	12	0	0	0	14
(m) Trojan:Win32/Matsnu	0	0	3	0	0	0	0	0	0	0	0	0	14	0	0	17
(n) TrojanDownloader:Win32/Dofoil	0	0	1	0	0	0	0	0	0	0	1	0	0	1	0	3
(o) TrojanSpy:Win32/Skeeyah	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Sum	281	178	147	74	70	40	40	30	26	25	24	14	14	10	10	983

(b) RF

	(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)	(i)	(j)	(k)	(l)	(m)	(n)	(o)	Sum
(a) TrojanSpy:Win32/Ursnif	274	0	27	0	12	0	0	0	0	0	0	0	0	0	1	314
(b) Worm:Win32/Gamarue	0	171	6	0	8	0	0	0	1	0	15	0	0	0	0	201
(c) Trojan:Win32/Dynamer	7	1	87	0	10	0	1	0	1	2	1	0	1	2	4	117
(d) TrojanDropper:Win32/Rovnix	0	0	1	74	5	0	0	0	0	0	0	0	0	0	0	80
(e) Trojan:Win32/Skeeyah	0	1	15	0	27	0	2	0	0	2	1	2	0	0	3	53
(f) Ransom:Win32/Crowti	0	0	2	0	1	40	0	0	0	0	1	0	0	0	0	44
(g) Trojan:Win32/Tinba	0	0	0	0	3	0	37	0	0	0	2	0	0	0	0	42
(h) Ransom:Win32/Teerac	0	0	1	0	0	0	0	30	0	0	1	0	0	0	0	32
(i) Ransom:Win32/Isda	0	3	0	0	1	0	0	0	23	0	2	0	0	0	0	29
(j) PWS:Win32/Fareit	0	0	1	0	2	0	0	0	0	21	0	0	0	0	0	24
(k) Trojan:Win32/Bagsu	0	2	0	0	0	0	0	0	1	0	0	0	0	0	0	3
(l) Trojan:Win32/Avkill	0	0	0	0	1	0	0	0	0	0	0	12	0	0	0	13
(m) Trojan:Win32/Matsnu	0	0	3	0	0	0	0	0	0	0	0	0	13	0	0	16
(n) TrojanDownloader:Win32/Dofoil	0	0	2	0	0	0	0	0	0	0	1	0	0	8	0	11
(o) TrojanSpy:Win32/Skeeyah	0	0	2	0	0	0	0	0	0	0	0	0	0	0	2	4
Sum	281	178	147	74	70	40	40	30	26	25	24	14	14	10	10	983

(c) NB

	(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)	(i)	(j)	(k)	(l)	(m)	(n)	(o)	Sum
(a) TrojanSpy:Win32/Ursnif	279	0	32	0	12	0	0	0	0	0	0	0	0	0	0	323
(b) Worm:Win32/Gamarue	0	31	2	0	3	0	0	0	0	0	3	0	0	0	0	39
(c) Trojan:Win32/Dynamer	0	0	28	0	2	0	0	0	0	0	0	0	0	1	0	31
(d) TrojanDropper:Win32/Rovnix	0	140	7	72	13	0	0	0	0	0	11	0	0	0	0	243
(e) Trojan:Win32/Skeeyah	0	0	11	0	0	0	0	0	0	1	0	0	0	0	0	12
(f) Ransom:Win32/Crowti	0	0	4	0	0	39	0	0	0	0	1	0	0	0	0	44
(g) Trojan:Win32/Tinba	0	0	1	0	5	0	39	0	0	0	2	0	0	0	0	47
(h) Ransom:Win32/Teerac	0	1	2	0	4	1	0	30	13	7	1	0	0	0	0	59
(i) Ransom:Win32/Isda	0	2	0	0	0	0	0	0	13	8	2	0	0	0	0	25
(j) PWS:Win32/Fareit	0	0	13	1	6	0	0	0	0	7	0	0	0	0	1	28
(k) Trojan:Win32/Bagsu	0	3	4	0	1	0	0	0	0	0	1	0	0	0	3	12
(l) Trojan:Win32/Avkill	0	0	1	0	4	0	0	0	0	1	0	14	0	0	1	21
(m) Trojan:Win32/Matsnu	0	0	3	0	1	0	0	0	0	0	0	0	14	0	0	18
(n) TrojanDownloader:Win32/Dofoil	0	1	11	1	3	0	1	0	0	0	2	0	0	9	0	28
(o) TrojanSpy:Win32/Skeeyah	2	0	28	0	16	0	0	0	0	1	1	0	0	0	5	53
Sum	281	178	147	74	70	40	40	30	26	25	24	14	14	10	10	983

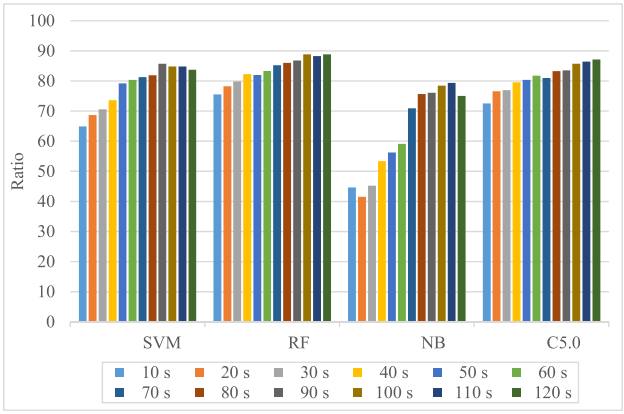
(d) C5.0

	(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)	(i)	(j)	(k)	(l)	(m)	(n)	(o)	Sum
(a) TrojanSpy:Win32/Ursnif	277	0	28	0	10	0	0	0	0	0	0	0	0	0	1	316
(b) Worm:Win32/Gamarue	0	169	6	0	8	0	0	0	0	0	15	0	0	0	0	198
(c) Trojan:Win32/Dynamer	4	1	80	1	17	0	1	0	0	1	1	0	2	3	6	117
(d) TrojanDropper:Win32/Rovnix	0	0	2	73	5	0	0	0	0	0	0	0	0	0	0	80
(e) Trojan:Win32/Skeeyah	0	2	13	0	18	0	0	0	1	3	0	0	0	0	2	39
(f) Ransom:Win32/Crowti	0	0	1	0	0	40	0	0	0	0	1	0	0	0	1	43
(g) Trojan:Win32/Tinba	0	0	0	0	5	0	39	0	0	0	2	0	0	0	0	46
(h) Ransom:Win32/Teerac	0	0	3	0	0	0	0	30	0	0	1	0	0	0	0	34
(i) Ransom:Win32/Isda	0	3	1	0	2	0	0	0	25	0	2	0	0	0	0	33
(j) PWS:Win32/Fareit	0	0	4	0	2	0	0	0	0	20	0	0	0	0	0	26
(k) Trojan:Win32/Bagsu	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
(l) Trojan:Win32/Avkill	0	0	0	0	1	0	0	0	0	1	1	14	0	0	0	17
(m) Trojan:Win32/Matsnu	0	0	3	0	0	0	0	0	0	0	0	0	12	0	0	15
(n) TrojanDownloader:Win32/Dofoil	0	0	3	0	0	0	0	0	0	0	1	0	0	7	0	11
(o) TrojanSpy:Win32/Skeeyah	0	3	3	0	2	0	0	0	0	0	0	0	0	0	0	8
Sum	281	178	147	74	70	40	40	30	26	25	24	14	14	10	10	983

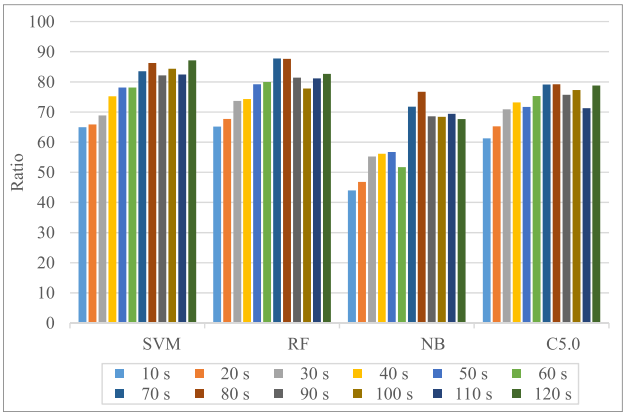
Table 9 Number of samples classified.

Threshold	10	20	30	40	50	60
# of samples	1,764	1,600	1,375	1,149	1,068	983

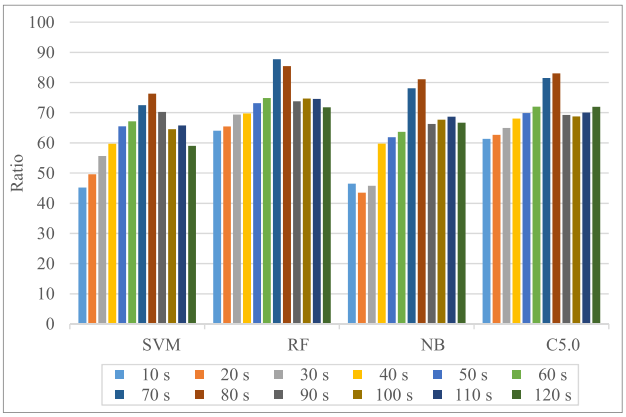
Threshold	70	80	90	100	110	120
# of samples	637	543	456	442	427	413



(a) Micro-averages of precision and recall



(b) Macro-averages of precision



(c) Macro-averages of recall

Fig. 3 Effects of varying the threshold of cumulative sleep time.

of varying the threshold tends to be large when the threshold is small, while the values tend to become stable when the threshold is large, particularly for thresholds larger than 90 s. Another trend is that the values gradually become better with increases in the threshold. Naturally, when the threshold becomes larger, the classification becomes easier because a large threshold causes target samples to include only extreme cases in terms of sleep behavior. We believe that a threshold of 60 s is a reasonable choice to limit the parameter space and conduct further tests in which other parameters are varied.

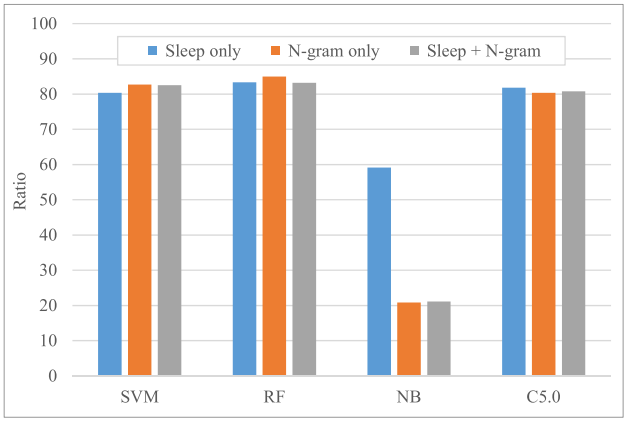
4.2.3 Comparison with Existing Method

We compared the ability to classify malware based on sleep-related information with an existing method that also learns past API call sequences and makes a judgment on unknown sequences. We chose the *N-gram* method as the comparison counterpart because it is well known, and a significant body of research has reported its effectiveness in malware classification and detection [2], [11], [41], [47], [48]. This method is also composed of a training phase and a classification phase. In the training phase, it creates a set of *N-grams* out of a given API call sequence. N-grams are subsequences of the original sequence whose lengths are N. A feature vector, consisting of bits, each of which is associated with a distinct N-gram created from all samples used in the training phase, is associated with each malware sample. A bit is set to 1 if the associated N-gram is found in the set of N-grams of the malware sample, and 0 otherwise. This method provides learning algorithms with pairs of a feature vector and its malware name in the Microsoft classification as the training data. It then creates models to label feature vectors with malware names. In the classification phase, it labels unknown feature vectors with predicted malware names using the models. As done in experiments described in previous work [25], we combine consecutive calls of the same API function into one call.

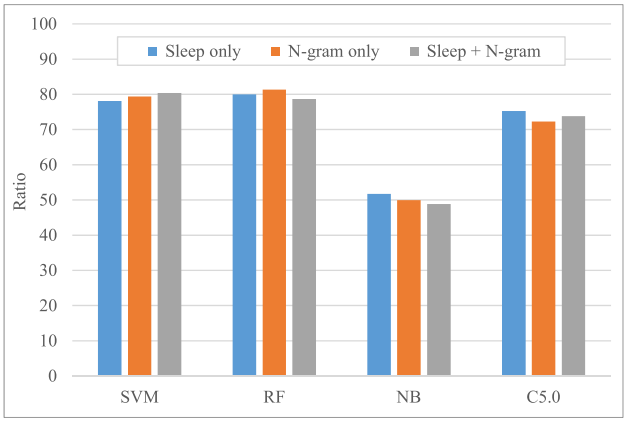
In addition to the sleep-based method and the N-gram-based one, we implemented a third method, which is the integration of these two methods. This method associates each malware sample with a feature vector that is a concatenation of the feature vector of the sleep-based method and the feature vector of the N-gram method.

As described in Section 1, we do not intend to substitute sleep behavior alone for rich information sources such as file I/O and registry key accesses. Instead, we consider that it is critical to integrate the findings of sleep studies with existing methods. Because a sandbox can easily monitor resource accesses, malware developers have sufficient motivation to suppress or disguise them for evasion. Therefore, we consider that exploring new clues for malware classification and detection is extremely important. We introduce the integration method as an example of methods that integrate sleep information with well-established sources of information for malware detection.

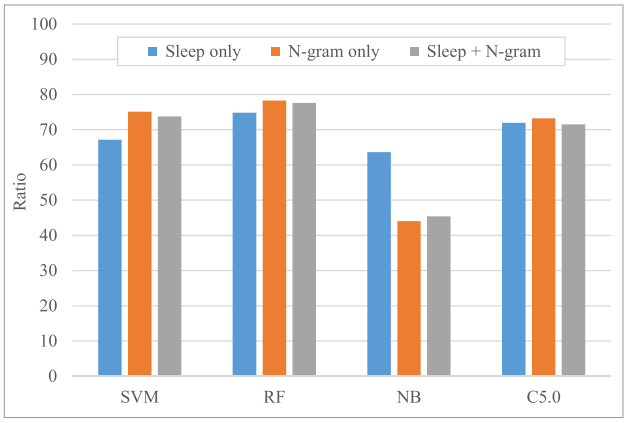
We compared the abovementioned three methods in cross-validation tests, whose configuration is the same as the first test (60 s threshold and 983 samples). We varied the value of N between one, two, and three. In this paper, however, we present the results obtained when N was set to three, because the overall trends were the same for the three configurations. **Figure 4** shows the comparison of the three methods. The labels "Sleep only," "N-gram only," and "Sleep + N-gram" in the figures represent the sleep-based, N-gram, and integrated methods, respectively. The difference in precision and recall values between the methods was small in SVM, RF, and C5.0. When using NB, the N-gram method and integration method performed significantly worse than the sleep method. We cannot determine the



(a) Micro-averages of precision and recall



(b) Macro-averages of precision



(c) Macro-averages of recall

Fig. 4 Comparison with the N-gram method.

best method from these results alone because no method consistently outperformed the others. The sleep-based method was the best in some cases, the N-gram-based method was the best in some cases, and the integration method was best in the other cases. We emphasize here that the sleep-based method could exceed an existing method in certain cases, and that the integration of sleep information into an existing method could also improve the classification ability in certain cases. In addition, sleep information alone provided classification ability comparable to a set of N-grams that includes all subsequences of a whole API call sequence.

4.2.4 Reasons for Classification Errors

The classification errors are caused by a complex combination of reasons, and for now we refrain from identifying the primary reason. Instead, we would like to mention that one probable reason is that sleep behavior differs significantly between some samples within the same malware family. We explain this with an example. **Table 10** lists the partial feature values of 15 out of 147 Trojan:Win32/Dynamer samples. Although all have the same malware name, their feature values are quite diverse. Moreover, these samples behave differently in terms of accessed files, registry keys, and network addresses. For example, no pair of samples reads the same set of registry keys in the execution of the main process. It is arguable whether it is reasonable and/or beneficial to classify these samples into the same family. However, prudent judgment is required on this matter, because there may be a class of malware that prepares multiple patterns of operation and randomly chooses one of them at runtime.

More validation is needed to understand whether it is appropriate to completely trust the labeling assigned by the Microsoft product and consider this the ground truth. In terms of API call sequences, samples that behave quite similarly have different malware names, and the sleep behavior of samples with the same malware name differs significantly. The same phenomena can be observed in the labeling given by other anti-malware products, and we have not found one whose labeling seems entirely reasonable.

In addition, there is considerable dispersion between the labeling assigned by different products. For example, a Symantec product further classifies 281 samples of TrojanSpy:Win32/Ursnif into seven families. The selection and/or composition of "ground-truth data" from multiple anti-malware products is an essential problem that has been extensively studied [3], [28], [39].

4.2.5 Case Study: C5.0 Decision Trees

Finally, we briefly explain which features were regarded as effective by the learning algorithms. We adopt C5.0 as a sample case. C5.0 creates decision trees for classification, and these provide a clue as to the relative importance of each feature.

Figure 5 shows the decision tree displayed by the C5.0 program using the whole training data (1,087 vectors of feature values). Equality and inequality expressions in the figure are branch conditions. When classifying samples, the user follows the tree from the root node to a leaf node, branching according to the conditions. Each leaf node is associated with an abbreviated malware name, which is assigned to samples classified into the leaf. The numbers in the parentheses after the malware names represent the result of the training-data test, which the C5.0 program performs by default. In the test, C5.0 first creates a tree from the given training data, and then classifies the training data, providing the data again to the resulting tree. The training-data test supports users to estimate how precisely the tree can classify data that are similar to the training data. The numbers on the left-hand side in the parentheses indicate the number of samples classified into the corresponding leaf, whereas the numbers on the right-hand side indicate the number of incorrectly classified samples. The decision tree in Fig. 5 classifies the given samples into 34 malware families using 22 features. It does not classify samples into the remaining 33 malware families that are not associated with any

Table 10 Feature values of some Trojan:Win32/Dynamer!ac samples.

(1)	(2)	(3)	(4)	(5)	(11)	(13)	(14)	(15)	(16)	(17)	(18)	(19)	(22)	(23)	(24)	(25)
2,728,163,227	0	209,858,879	20	4	13	2	4	9	0.25	1	0.16	1	.	.	true	.
60,000	60,000	60,000	60,000	1	1	1	1	1	0	669	0	0	.	.	.	.
60,000	60	189.4	60	6	1,009	4	1,009	1	0	2	0.5	0	true	true	.	true
30,000	2,000	24,400	30,000	2	5	3	5	1	0	67	0.01	0	.	.	.	true
30,000	0	1,334.6	0	88	147	6	62	41	0.91	1	1	0	.	.	true	.
25,000	2,500	3,365.4	2,500	2	26	1	26	1	0	342	0.02	0	true	.	.	.
20,000	4	267.3	100	11	601	5	21	135	0.18	2	0.99	1	true	.	true	true
14,835	6	1,011.7	12	91	217	5	12	124	0.9	2	0.9	0	true	.	true	.
10,000	1,000	3,185.2	1,000	4	27	2	27	1	0	17	0.33	0	.	true	true	.
5,000	50	128.5	50	4	1,485	4	1,485	1	0	17	0.05	1	true	.	.	.
3,000	3,000	3,000	3,000	1	27	1	27	1	0	7,034	0	0	.	.	.	.
3,000	0	119.3	0	3	1,433	102	590	4	0	1	0.24	0	true	.	true	.
600	100	342.7	100	3	199	2	6	100	0	38	0.82	0	.	.	.	.
181	181	181	181	1	465	1	465	1	1	169	0.42	0	.	true	.	.
5	5	5	5	1	14,986	1	1	14,986	1	188	0.99	0	.	.	.	.

```
ntimekinds > 4:
:...nprocs <= 2:
:   :...maxsleepratio <= 0.46:
:   :   :...progression > 0:
:   :   :   :...nthreads <= 4: Kasidet (3)
:   :   :   :   nthreads > 4: Neurevt (3/2)
:   :   :   progression <= 0: ... (omitted)
:   :   maxsleepratio > 0.46:
:   :   :...unroundedratio > 0.5:
:   :       :...thread <= 0: Necurs (3/1)
:   :       :   thread > 0: Dynamer (64/6)
:   :       unroundedratio <= 0.5:
:   :       :...fileattr <= 0: TrojanSkeeyah (4/2)
:   :           fileattr > 0: ... (omitted)
:   nprocs > 2:
:   :...nthreads > 9: Tinba (47/8)
:       nthreads <= 9:
:       :...unroundedratio > 0.25: Matsnu (15/1)
:           unroundedratio <= 0.25:
:           :...maxchunktime > 38360: Dynamer (6/1)
:               maxchunktime <= 38360: ... (omitted)
ntimekinds <= 4:
:...maxtime > 37000:
    :...earliestsleep > 195: Gamarue (176/39)
    :   earliestsleep <= 195:
    :   :...minchunktime > 30000: Rovnix (68/7)
    :       minchunktime <= 30000: ... (omitted)
    :       :...maxsleepratio > 0.19: Rovnix (15/2)
    :           maxsleepratio <= 0.19: ... (omitted)
    maxtime <= 37000:
    :...nprocs > 1:
        :...nprocs > 2:
        :   :...mintime <= 5: Crowti (16/5)
        :   :   mintime > 5: ... (omitted)
        :   nprocs <= 2: ... (omitted)
        nprocs <= 1:
        :...maxsleepratio <= 0.02:
            :...maxchunksize > 16: Avkill (5/1)
            :   maxchunksize <= 16: ... (omitted)
            maxsleepratio > 0.02: ... (omitted)
```

Fig. 5 Decision tree created by C5.0 algorithm (the output is partially omitted, and the format is partially modified).

leaf node.

We measured the usage proportion of each feature for the tree in Fig. 5 and four decision trees created from partial training data in the cross-validation test. In the four subtests of the cross-validation test, three quarters of the data for the test are provided to the C5.0 algorithm to create a decision tree. **Table 11** lists those features whose usage proportions for the tree in Fig. 5 are more than 10%. The top row indicates the training data used to create the decision tree. Partial data 1, 2, 3, and 4 indicate training data used in each subtest of the cross-validation test. The C5.0 algorithm judges these feature values as being effective for classification. As is obvious from the values in the table, the usage of features strongly depends on the given data and differs significantly between trees. However, we can still find several features that are used to some extent in all trees. Such features are nprocs, maxsleepratio, maxtime, fileattr, and earliestsleep, which exhibit proportions higher than 10% in all trees.

It cannot be concluded from these results that the usage trends presented here are universal. The FFRI Dataset is not large and the numbers of samples of each malware family vary significantly. Hence, we expect readers to be aware of the dependency on the FFRI Dataset and the potential impacts caused by the bias in the numbers of samples. In future work, it would be important to conduct further measurements with a larger sample set and evaluate the generality of the obtained results.

4.3 Knowledge Obtained from Individual API Call Sequences

4.3.1 Purposes of Sleep

We surmise that many malware samples executed sleeps to avoid an excessive load increase. One type of behavior observed in many samples is sleeping for a while (typically from several milliseconds to several seconds) between repeated operations. For example, many samples slept between successive queries to a single DNS server. In addition, many samples also slept between the resumption and suspension of another thread, between successive scans of a process list, and between successive checks of file attributes. For example, **Figure 6** shows a part of the call sequence of a Trojan:Win32/Tinba.F sample, which sleeps for a second between every 401 DNS queries.

Some malware threads simply repeat the invocation of `NtDelayExecution` and do not execute any other operation. Many others sleep for a long time at the early stage of their execution. Some others sleep for several seconds between multiple acquisition of the cursor position and foreground window. We surmise that the purpose of sleeps in these threads is not load reduction, but rather some malware-specific aim such as anti-analysis and delitescence. A part of the sleeps may be to detect an analysis or sandbox system through user behavior measurement or a combination of sleeps and elapsed-time measurement. Obviously, further research is needed to clarify the purpose of sleeps in

Table 11 Usage proportions of features in decision trees.

	Whole training data	Partial data 1	Partial data 2	Partial data 3	Partial data 4
(5) ntimekinds	100.0%	15.3%	5.3%	32.1%	100.0%
(12) nprocs	72.3%	58.9%	53.1%	53.3%	70.2%
(18) maxsleepratio	63.1%	11.7%	17.3%	16.9%	17.6%
(1) maxtime	52.4%	43.5%	52.0%	50.4%	51.6%
(16) unroundratio	42.3%	0%	0%	2.2%	4.6%
(25) fileattr	32.2%	73.9%	55.7%	42.4%	42.2%
(17) earliestsleep	31.9%	26.2%	40.7%	100.0%	34.0%
(15) maxchunksize	17.4%	35.8%	5.4%	0%	4.5%
(2) mintime	15.7%	3.0%	36.6%	0%	16.9%
(14) nchunks	15.3%	0%	0.7%	0%	0%
(7) minchunktime	11.5%	0%	20.9%	12.6%	12.4%

```
gethostbyname("eeqognneyyeu.in") = WSAHOST_NOT_FOUND
gethostbyname("eeqognneyyeu.ru") = WSAHOST_NOT_FOUND
NtDelayExecution(1000) = 0
gethostbyname("g0jdy3826yenz63om.cc")
                                 = WSAHOST_NOT_FOUND
gethostbyname("ijjnehxchgde.com") = WSAHOST_NOT_FOUND
gethostbyname("ijjnehxchgde.net") = WSAHOST_NOT_FOUND
... (396 calls of gethostbyname)
gethostbyname("eeqognneyyeu.in") = WSAHOST_NOT_FOUND
gethostbyname("eeqognneyyeu.ru") = WSAHOST_NOT_FOUND
NtDelayExecution(1000) = 0
gethostbyname("g0jdy3826yenz63om.cc")
                                 = WSAHOST_NOT_FOUND
gethostbyname("ijjnehxchgde.com") = WSAHOST_NOT_FOUND
gethostbyname("ijjnehxchgde.net") = WSAHOST_NOT_FOUND
```

Fig. 6 Sleeps between DNS queries.

```
NtQueryDirectoryFile("C:\\Windows", ...) = 0
NtQueryDirectoryFile("C:\\Windows", ...)
                                    = STATUS_NO_MORE_FILES
NtClose(...) = 0
NtDelayExecution(0) = STATUS_NO_YIELD_PERFORMED
NtDelayExecution(3) = 0
NtDelayExecution(6) = 0
... (10 calls of NtDelayExecution with arithmetic sequence numbers)
NtDelayExecution(39) = 0
NtDelayExecution(42) = 0
NtDelayExecution(45) = 0
CreateDirectoryW("C:\\...\\Roaming\\alFSVWJB\\") = 1
```

Fig. 7 Sequence of sleeps with arithmetic sequence times.

```
NtResumeThread(...) = 0
NtClose(...) = 0
NtClose(...) = 0
NtDelayExecution(162) = 0
NtDelayExecution(148) = 0
NtDelayExecution(135) = 0
NtDelayExecution(205) = 0
... (24 calls of NtDelayExecution with seemingly random times)
NtDelayExecution(233) = 0
NtDelayExecution(215) = 0
NtDelayExecution(153) = 0
NtDelayExecution(224) = 0
CreateThread(...) = ...
```

Fig. 8 Sequence of sleeps with seemingly random times.

more detail. Unfortunately, doing this with just the FFRI Dataset is difficult because the dataset does not contain logs of instruction executions or executable programs.

4.3.2 Choice of Sleep Times

The preference for sleep times varies among the malware. Because `NtDelayExecution` calls with unrounded sleep times constitute only a small proportion of all `NtDelayExecution` calls for most malware samples, such invocations stand out somewhat.

Some malware sleep for a fixed amount of time, whereas other programs sleep for different amounts of time. Some malware gradually increase their sleep time by small increments, whereas others sleep for various seemingly random times. **Figure 7** shows a part of the call sequence of a sample labeled Backdoor:Win32/Kasidet.C by Microsoft. The sample logs contain a chunk with sleep times of multiples of three beginning from zero. The same chunk appears in the logs of the other three Kasidet samples, one Neurevt sample, and one Skeeyah sample.

Figure 8 shows a part of the call sequence of a sample labeled TrojanDownloader:Win32/Silcon!rfn by Microsoft, which includes a chunk composed of seemingly random sleep times within a small range. This chunk appears in the logs of many other samples.

We found that sleep times were strongly related to their locations in call sequences. Inherent sleep times or inherent types of sleep times clump in different parts of an API call sequence. For example, the Silcon sample described above seems to have several types of sleep times that appear in clumps, including (1) 28–96 ms random sleeps between `NtResumeThread` and `NtSuspendThread`, (2) a chunk of 125–248 ms random sleeps, (3) a chunk of 1,012–2,915 ms random sleeps lasting until the analysis time-out, (4) a chunk of 4,133–11,913 ms random sleeps lasting until the analysis time-out, (5) a 5-ms sleep as the only API call by a certain thread, and (6) 0-ms sleeps after call sequences for DNS communication. It is likely that the types of sleep times are related to the execution phases of a program. Although it seems interesting to extend this study by taking execution phases into account, we expect that this will be another challenging task.

4.3.3 Use of Many Sleeping Threads

Characteristic sleep behavior was observed in the call sequences of several Trojan:Win32/Skeeyah.A!rfn samples. The samples provide a string including "`Themida Professional ... (c)2010 Oreans Technologies`" to an API call argument, and are expected to be protected by the Themida Protector [32]. Themida transforms a given binary program into another binary program that behaves in the same way, but executes many sophisticated protection operations. Below, we describe the behavior of one of the samples in detail. The sample creates one child process. The parent process is composed of 30 threads, 26 of which

call NtDelayExecution. The child process is composed of 27 threads, 25 of which call NtDelayExecution. In the execution of 41 of the 30 + 27 = 57 threads, the call sequence is composed of only NtDelayExecution, and most of the sleep times are 0 or 2,001 ms. The sample calls timeGetTime, which returns the current system time, immediately before and after CreateThread calls for thread creation. We expect that this sample is attempting to detect an analysis system using time information. However, we have not obtained any direct evidence. Although such "aggressive" sleeps complicate sleep-conscious static or dynamic analyses, we consider that this actually facilitates malware classification and detection because of its inherent characteristic.

5. Discussion

5.1 Awareness of Sleep Behavior Analysis

If a security mechanism using sleep behavior is implemented for malware detection or classification, the malware creator may become aware of the mechanism and attempt to evade it by crafting stealthy sleeps. The problem of attackers' being aware of protection-side mechanisms is not specific to malware classification using sleep behavior, but is a universal and traditional issue in the field of malware classification or detection [7], [14], [22], [34], [46], [49]. In this work, we would like to concentrate on clarifying what the current state-of-the-art can achieve and rotate the cycle of attack and protection technology.

5.2 Origin of Sleeps

Not all malware behaviors are caused by the source program itself; some are instead often caused by supportive software. A considerable proportion of modern malware are protected with a packer, which introduces additional behavior to the malware's execution. In addition, reusable code such as libraries and malware creation kits are widely used to develop malware. For example, among the 1,234 samples, we found that at least 12 samples are likely to be protected by Themida [32] and at least 12 other samples by VMProtect [44]. In terms of sleep behavior, it is vital to consider whether the sleep behavior is caused by the main malware program or by supportive software. The origin of the sleep behavior has not been considered in this study, because it is difficult to obtain or estimate such information from the dataset alone. As a result, this study may capture both the features attributed to the main malware program itself and those attributed to supportive software.

We examined the classification of the Themida and VMProtect samples. Five of the Themida samples and two of the VMProtect samples were excluded because they were not labeled in the Microsoft classification. Another Themida sample was further excluded from the cross-validation tests because of the number of samples in the family. The sleep-based method described in Section 4.2.1 with the RF algorithm correctly classified two of the remaining six Themida samples and all of the remaining 10 VMProtect samples. However, we have not identified how strongly the behavior of these packers affected this result.

We must carefully consider whether to ignore sleeps initiated by supportive software, because the choice of supportive software is likely to be closely related to the malware family. Therefore, it is not certain at present whether the values of the classification obtained in our measurement would rise or fall if we could successfully ignore sleeps initiated by supportive software. We believe that identification of packers is as important and essential as identification of malware families. In future work, it is necessary to extend the analysis by taking the origin information into consideration based on the binaries of malware programs.

```
static long winMutex_lock = 0;

static int winMutexInit(void){
  /* The first to increment to 1 does actual
     initialization */
  if( InterlockedCompareExchange(&winMutex_lock, 1, 0)
      ==0 ){
    ...
    winMutex_isInit = 1;
  }else{
    /* Someone else is in the process of initing the
       static mutexes */
    while( !winMutex_isInit ){
      Sleep(1);
    }
  }
  return SQLITE_OK;
}
```

Fig. 9 Example of sleep calls in a loop.

```
VOID timing_sleep_loop (UINT delay)
{
  ...
  int delay_divided  = delay / 1000;
  ...
  for (int i = 0; i < 1000; i++) {
    Sleep(delay_divided);
  }
  ...
}
```

Fig. 10 Division of a long sleep into many short sleeps in an anti-sandbox operation.

5.3 Reasons for Consecutive Short Sleeps

Table 3 shows that the target samples called NtDelayExecution 600.8 times on average and generated 60.2 chunks on average. Although the numbers may seem abnormally large, we are aware of sufficient reasons for them being large. A major reason is that NtDelayExecution is often called in a loop body. Typical purposes of such calls include waiting for a short period between operations that poll events such as network communication, termination of services, process spawns, lock releases, and file handle releases. Sleeps in a loop constitute a chunk if the polling operations do not cause other API calls. Another typical purpose is throttling program execution to avoid excessive loads. These usages of sleeps are also found abundantly among benign software. An example is shown in **Fig. 9**, which shows a source code fragment of Firefox in which a sleep API function is repeatedly called between polling operations. In the program part, Firefox waits for another thread to initialize mutexes, executing 1 ms sleeps repeatedly between checks.

Malware have additional reasons to execute short sleeps in a loop. **Figure 10** is a slightly modified source code fragment of

Al-Khaser [1], which is a sandbox detection tool that executes numerous anti-sandbox operations. The code attempts to detect a sandbox by executing long sleeps and observing whether the sleeps are actually executed or skipped. As shown in the code, a long sleep is achieved with 1,000 consecutive short sleeps. A comment in the source code describes that such division can induce sandboxes to refrain from skipping sleeps because skipping can lead to race conditions and short sleeps are just negligible. Using such divisions in sleep periods, malware can conceal long sleeps and appear relatively harmless to sandboxes.

Although this study has not thoroughly investigated the purpose of each sleep, it is an important aspect to be studied, and future investigations on this can be expected to provide fruitful insights.

6. Related Studies

Many sandbox systems are equipped with countermeasures against anti-analysis sleep behavior [17], [27], [29], [35]. These sandboxes execute "anti-anti-analysis" operations such as skipping sleeps by malware and controlling the speed of virtual time. Such systems reduce the effects of anti-analysis operations, whereas our study attempted to better understand the sleep behavior and explore a classification technology for malware that aggressively leverages the sleep behavior diversity.

There have been many studies on malware classification techniques in which a *similarity* or *distance* between the behavior of multiple malware samples is defined and leveraged for classification. Fujino et al. [19] proposed a method of malware classification targeted at the API call sequences in the FFRI Dataset. Their method first identifies an *API call topic* of each subset of malware samples, which is the information on API calls that frequently occur in the call sequences of the subset. Unknown malware samples are then classified according to the distance between the call sequence of the sample and the API call topic of each subset. Wagener et al. [45] classified malware samples based on a similarity between API call sequences calculated with a Hellinger distance matrix. Their method regards each API function as a symbol, and does not use information about the specification of each API function or the function arguments. Apel et al. [2] compared multiple schemes to define the distances between the behavior of malware samples. These studies differ from the present research in that they do not leverage the knowledge of the specification of individual API functions. Our study focuses on a specific API function for sleeps, and attempts to clarify various characteristics of behavior related to this function. The aforementioned studies are complementary to the work reported in this paper.

Gao et al. [20] conducted pioneering work on the notion of the behavioral distance between multiple system call sequences. Their work dealt with anomalous sequences of system calls that are mainly caused by software faults or attacks against benign programs. In contrast to our study, they did not investigate the behavioral distances between different malware samples.

Kolosnjaji et al. [25] classified malware samples through the deep learning of sequences of Windows API calls invoked by malware. Their method characterizes malware behavior with N-grams of call sequences composed of function name information. In addition, their method does not make use of knowledge in specifications for individual API functions.

Oyama [33] uncovered the proportion of malware samples executing anti-analysis operations and the trends of anti-analysis operations executed by a set of modern malware. Several other studies [5], [9], [12], [13] have also reported the proportion of malware samples that execute anti-analysis operations and categorized the operation types. In contrast to our study, these approaches only provide a bird's-eye view of trends in various anti-analysis operations and do not investigate the behavior of individual operations executed by malware.

Crandall et al. [15] proposed a system for building the "timetable" of malware behavior and identifying code fragments of time-dependent behavior. This system manipulates the time information observed by the malware in a virtual machine. The system enables malware analysts to recognize code fragments executed after a given amount of time without actually waiting for the passage of time. Their study develops a useful method for analyzing time-related operations executed by malware, whereas our study highlights the detailed temporal trends.

HASTEN [24] is a dynamic analysis system that reduces the effect of anti-analysis operations to delay the execution of the malware itself. Although this aspect is similar to our study, HASTEN primarily targets execution-delaying operations realized by the repeated execution of dummy operations such as dummy API calls, and does not target execution-delaying operations realized by sleeps.

7. Summary and Future Work

This paper reported the results of an investigation on the characteristics of malware sleep behavior. A wide variety of feature values were extracted, and we determined that sleep behavior is a promising source of features for distinguishing between different malware samples. We also classified malware samples that cumulatively sleep for a long time, based on their sleep behavior using four learning algorithms. The best algorithm, RF, achieved 83.3% micro-average precision, 80.0% macro-average precision, and 74.8% macro-average recall in a cross-validation test. Further, we presented a classification case study using C5.0, the results of which revealed that the features that were frequently used in the decision trees were (1) the number of processes, (2) the maximum ratio of `NtDelayExecution` calls in a call sequence, (3) the maximum sleep time, (4) whether a fileattr-related API function is called before or after a sleep call, and (5) earliest call position of `NtDelayExecution`. However, we also find that the choice of features is affected by training data, and we do not conclude that these features are the most useful in every case.

There are several directions for future work. First, it is necessary to further investigate the behavior of those samples for which the learning algorithms failed to assign the correct malware names, and clarify the reason for these incorrect determinations. Second, it is also desirable to identify or estimate the purpose of many sleep calls, because we do not yet fully understand why individual sleep calls are used at certain times by malware. We believe that more accurate knowledge of these purposes will improve the design of the feature set. Finally, it would

be interesting to extend this study by taking into account stealth sleeps achieved without `NtDelayExecution`, such as a combination of time measurement and dummy operations. We anticipate that finding stealth sleeps from API call sequences is a challenging task that will require a novel technique.

Acknowledgments We are grateful to Yosuke Chubachi, Hirotaka Kokubo, and Kensuke Takahashi for useful comments. We would like to extend our gratitude to FFRI, Inc. and the MWS organizing committee for providing FFRI Dataset. This work was supported by JSPS KAKENHI Grant Numbers 26330080 and 17K00179.

References

[1] Al-Khaser, available from ⟨https://github.com/LordNoteworthy/al-khaser/⟩.
[2] Apel, M., Bockermann, C. and Meier, M.: Measuring Similarity of Malware Behavior, *Proc. 5th LCN Workshop on Security in Communications Networks*, pp.891–898 (2009).
[3] Bailey, M., Oberheide, J., Andersen, J., Mao, Z.M., Jahanian, F. and Nazario, J.: Automated Classification and Analysis of Internet Malware, *Proc. 10th International Symposium on Recent Advances in Intrusion Detection*, pp.178–197 (2007).
[4] Barabosch, T., Dombeck, A., Yakdan, K. and Gerhards-Padilla, E.: BotWatcher - Transparent and Generic Botnet Tracking, *Proc. 18th International Symposium on Research in Attacks, Intrusions, and Defenses*, pp.565–587 (2015).
[5] Barbosa, G.N. and Branco, R.R.: Prevalent Characteristics in Modern Malware, *Black Hat USA 2014* (2014).
[6] Benoit, M.: Mitigating Cybersecurity Risk with Palo Alto Networks and Splunk, *.conf 2013* (2013), available from ⟨https://conf.splunk.com/speakers/2013.html⟩.
[7] Blackthorne, J., Bulazel, A., Fasano, A., Biernat, P. and Yener, B.: AVLeak: Fingerprinting Antivirus Emulators through Black-Box Testing, *Proc. 10th USENIX Workshop on Offensive Technologies* (2016).
[8] Boser, B.E., Guyon, I.M. and Vapnik, V.N.: A Training Algorithm for Optimal Margin Classifiers, *Proc. 5th Annual Workshop on Computational Learning Theory*, pp.144–152 (1992).
[9] Branco, R.R., Barbosa, G.N. and Neto, P.D.: Scientific but Not Academical Overview of Malware Anti-Debugging, Anti-Disassembly and Anti-VM Technologies, *Black Hat USA 2012* (2012).
[10] Breiman, L.: Random Forests, *Machine Learning*, Vol.45, pp.5–32 (2001).
[11] Canali, D., Lanzi, A., Balzarotti, D., Kruegel, C., Christodorescu, M. and Kirda, E.: A Quantitative Study of Accuracy in System Call-Based Malware Detection, *Proc. 2012 International Symposium on Software Testing and Analysis*, pp.122–132 (2012).
[12] Chen, P., Huygens, C., Desmet, L. and Joosen, W.: Advanced or Not? A Comparative Study of the Use of Anti-debugging and Anti-VM Techniques in Generic and Targeted Malware, *Proc. 31st IFIP International Conference on ICT Systems Security and Privacy Protection*, pp.323–336 (2016).
[13] Chubachi, Y. and Aiko, K.: SLIME: Automated Anti-Sandboxing Disarmament System, *Black Hat Asia 2015* (2015).
[14] Corona, I., Giacinto, G. and Roli, F.: Adversarial attacks against intrusion detection systems: Taxonomy, solutions and open issues, *Inf. Sci.*, Vol.239, pp.201–225 (2013).
[15] Crandall, J.R., Wassermann, G., de Oliveira, D.A.S., Su, Z., Wu, S.F. and Chong, F.T.: Temporal Search: Detecting Hidden Malware Timebombs with Virtual Machines, *Proc. 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp.25–36 (2006).
[16] Cristianini, N. and Shawe-Taylor, J.: *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*, Cambridge University Press (2000).
[17] Cuckoo Sandbox, available from ⟨https://cuckoosandbox.org/⟩.
[18] Domingos, P. and Pazzani, M.: On the Optimality of the Simple Bayesian Classifier under Zero-One Loss, *Machine Learning*, Vol.29, pp.103–130 (1997).
[19] Fujino, A., Murakami, J. and Mori, T.: Discovering Similar Malware Samples Using API Call Topics, *Proc. 12th Annual IEEE Consumer Communications and Networking Conference*, pp.140–147 (2015).
[20] Gao, D., Reiter, M.K. and Song, D.: Behavioral Distance for Intrusion Detection, *Proc. 8th International Symposium on Recent Advances in Intrusion Detection*, pp.63–81 (2005).
[21] Hatada, M., Akiyama, M., Matsuki, T. and Kasama, T.: Empowering Anti-malware Research in Japan by Sharing the MWS Datasets, *J. Inf. Process.*, Vol.23, No.5, pp.579–588 (2015).
[22] Huang, L., Joseph, A.D., Nelson, B., Rubinstein, B.I.P. and Tygar, J.D.: Adversarial Machine Learning, *Proc. 4th ACM Workshop on Security and Artificial Intelligence*, pp.43–58 (2011).
[23] Kirat, D., Vigna, G. and Kruegel, C.: BareCloud: Bare-metal Analysis-based Evasive Malware Detection, *Proc. 23rd USENIX Security Symposium*, pp.287–301 (2014).
[24] Kolbitsch, C., Kirda, E. and Kruegel, C.: The Power of Procrastination: Detection and Mitigation of Execution-Stalling Malicious Code, *Proc. 18th ACM Conference on Computer and Communications Security*, pp.285–296 (2011).
[25] Kolosnjaji, B., Zarras, A., Webster, G.D. and Eckert, C.: Deep Learning for Classification of Malware System Call Sequences, *Proc. 29th Australasian Joint Conference on Artificial Intelligence*, pp.137–149 (2016).
[26] Kruegel, C.: Evasive Malware Exposed and Deconstructed, *RSA Conference 2015* (2015), available from ⟨https://www.rsaconference.com/events/us15/agenda/sessions/2022/evasive-malware-exposed-and-deconstructed⟩.
[27] Lastline Labs: Not so fast my friend - Using Inverted Timing Attacks to Bypass Dynamic Analysis (2014), available from ⟨https://www.lastline.com/labsblog/not-so-fast-my-friend-using-inverted-timing-attacks-to-bypass-dynamic-analysis/⟩.
[28] Li, P., Liu, L., Gao, D. and Reiter, M.K.: On Challenges in Evaluating Malware Clustering, *Proc. 13th International Symposium on Recent Advances in Intrusion Detection*, pp.238–255 (2010).
[29] Lin, C.-H., Pao, H.-K. and Liao, J.-W.: Efficient dynamic malware analysis using virtual time control mechanics, *Computers & Security*, Vol.73, pp.359–373 (2018).
[30] LiŢă, C.V., Cosovan, D. and GavriluŢ, D.: Anti-emulation trends in modern packers: A survey on the evolution of anti-emulation techniques in UPA packers, *Journal of Computer Virology and Hacking Techniques*, Vol.14, No.2, pp.107–126 (2018).
[31] Microsoft: Naming malware, available from ⟨https://www.microsoft.com/en-us/wdsi/help/malware-naming⟩.
[32] Oreans Technologies: Themida, available from ⟨http://www.oreans.com/themida.php⟩.
[33] Oyama, Y.: Trends of anti-analysis operations of malwares observed in API call logs, *Journal of Computer Virology and Hacking Techniques*, Vol.14, No.1, pp.69–85 (2018).
[34] Parampalliy, C., Sekar, R. and Johnson, R.: A Practical Mimicry Attack Against Powerful System-Call Monitors, *Proc. 2008 ACM Symposium on Information, Computer and Communications Security*, pp.156–167 (2008).
[35] Payload Security: VxStream Sandbox, available from ⟨https://www.payload-security.com/products/vxstream-sandbox⟩.
[36] Quinlan, J.R.: Improved Use of Continuous Attributes in C4.5, *Journal of Artificial Intelligence Research*, Vol.4, No.1, pp.77–90 (1996).
[37] Quinlan, J.R.: *C4.5: Programs for Machine Learning*, Morgan Kaufmann (1992).
[38] Rish, I.: An empirical study of the naive Bayes classifier, Technical report, IBM (2001).
[39] Sebastián, M., Rivera, R., Kotzias, P. and Caballero, J.: AVclass: A Tool for Massive Malware Labeling, *Proc. 19th International Symposium on Research in Attacks, Intrusions, and Defenses*, pp.230–253 (2016).
[40] Symantec: Internet Security Threat Report, Vol.21, April 2016 (2016).
[41] Tan, K.M.C. and Maxion, R.A.: "Why 6?" Defining the Operational Limits of stide, an Anomaly-Based Intrusion Detector, *Proc. 2002 IEEE Symposium on Security and Privacy*, pp.188–201 (2002).
[42] Trend Micro: How can Advanced Sandboxing Techniques Thwart Elusive Malware?, available from ⟨https://www.trendmicro.com/vinfo/us/security/news/security-technology/how-can-advanced-sandboxing-techniques-thwart-elusive-malware⟩.
[43] Trend Micro: New EMOTET Hijacks a Windows API, Evades Sandbox and Analysis, TrendLabs Security Intelligence Blog (2017), available from ⟨https://blog.trendmicro.com/trendlabs-security-intelligence/new-emotet-hijacks-windows-api-evades-sandbox-analysis/⟩.
[44] VMProtect Software: VMProtect, available from ⟨https://vmpsoft.com/products/vmprotect/⟩.
[45] Wagener, G., State, R. and Dulaunoy, A.: Malware behavior analysis, *Journal of Computer Virology*, Vol.4, No.4, pp.279–287 (2008).
[46] Wagner, D. and Soto, P.: Mimicry Attacks on Host-Based Intrusion Detection Systems, *Proc. 9th ACM Conference on Computer and Communications Security*, pp.255–264 (2002).
[47] Warrender, C., Forrest, S. and Pearlmutter, B.: Detecting Intrusions Using System Calls: Alternative Data Models, *Proc. 1999 IEEE Symposium on Security and Privacy*, pp.133–145 (1999).

[48] Wressnegger, C., Schwenk, G., Arp, D. and Rieck, K.: A Close Look on *n*-Grams in Intrusion Detection: Anomaly Detection vs. Classification, *Proc. 2013 ACM Workshop on Artificial Intelligence and Security*, pp.67–76 (2013).
[49] Yokoyama, A., Ishii, K., Tanabe, R., Papa, Y., Yoshioka, K., Matsumoto, T., Kasama, T., Inoue, D., Brengel, M., Backes, M. and Rossow, C.: SandPrint: Fingerprinting Malware Sandboxes to Provide Intelligence for Sandbox Evasion, *Proc. 19th International Symposium on Research in Attacks, Intrusions and Defenses*, pp.165–187 (2016).
[50] Yunakovsky, S.: Neutrino modification for POS-terminals, available from ⟨https://securelist.com/neutrino-modification-for-pos-terminals/78839/⟩.

Editor's Recommendation

This paper examines sleep behavior of malware and proposes a classifier based on the sleep features. While there have been many papers on malware analysis, the approach solely focusing on the sleep behavior is unique, and the experimental results are well discussed and promising. The paper has clear novelty and practicality, and thus is selected as a recommended paper.

(Masayuki Terada, Chief examiner of SIGCSEC)

Yoshihiro Oyama received a Ph.D. degree in information science from The University of Tokyo, Japan, in 2001. He was an associate professor at the University of Electro-Communications, Japan, from 2006 to 2016. He is currently an associate professor at University of Tsukuba, Japan, since 2016.