Trends of anti-analysis operations of malwares observed in API call logs

著者別名	大山 恵弘
journal or publication title	Journal of Computer Virology and Hacking Techniques
volume	14
number	1
page range	69-85
year	2017-02
権利	This is a pre-print of an article published in Journal of Computer Virology and Hacking Techniques. The final authenticated version is available online at: https://doi.org/10.1007/s11416-017-0290-x
URL	http://hdl.handle.net/2241/00151543

doi: 10.1007/s11416-017-0290-x

Journal of Computer Virology and Hacking Techniques manuscript No.
(will be inserted by the editor)

Trends of Anti-Analysis Operations of Malwares Observed in API Call Logs

Yoshihiro Oyama

Received: date / Accepted: date

Abstract Some malwares execute operations that determine whether they are running in an analysis environment created by monitoring software, such as debuggers, sandboxing systems, or virtual machine monitors, and if such an operation finds that the malware is running in an analysis environment, it terminates execution to prevent analysis. The existence of malwares that execute such operations (*anti-analysis* operations) is widely known. However, the knowledge acquired thus far, regarding what proportion of current malwares execute anti-analysis operations, what types of anti-analysis operations they execute, and how effectively such operations prevent analysis, is insufficient. In this study, we analyze FFRI Dataset, which is a dataset of dynamic malware analysis results, and clarify the trends in the anti-analysis operations executed by malware samples collected in 2016. Our findings revealed that, among 8243 malware samples, 856 (10.4%) samples executed at least one type of the 28 anti-analysis operations investigated in this study. We also found that, among the virtual machine monitors, VMware was the most commonly searched for by the malware samples.

Keywords Malware · virtualization · virtual machine monitors · hypervisors · anti-analysis · anti-virtualization

Y. Oyama
University of Tsukuba,
1-1-1 Tennodai, Tsukuba, Ibaraki, Japan
Tel.: +81-29-853-5191
Fax: +81-29-853-2983
E-mail: oyama@cc.tsukuba.ac.jp

1 Introduction

It is widely known that some malwares execute operations that make it difficult to analyze them. A major operation involves determining whether the malware is running in an analysis environment, such as a virtual machine or debugger, and to prevent execution of the original operation if the malware is found to be running in such an analysis environment. This study is concerned with *anti-analysis* operations, i.e., countermeasures adopted by malwares against analyzers.

Many studies have investigated anti-analysis operations and evasive malwares that behave in consideration of analysis environments. Thus, researchers have acquired a vast amount of knowledge on malware-side techniques for detecting the presence of analysis systems such as virtual machine monitors and debuggers. Further, it is known that a vulnerability in a virtual machine monitor can cause a malware running in a virtual machine to escape from it and execute arbitrary code in the host operating system [1]. Some studies have presented statistical data on real-world analysis-aware malwares. For example, in its April 2016 threat report, Symantec stated that approximately 16% of malwares are routinely able to detect and identify a virtual machine environment [2]. Methods for stealth analysis (i.e., methods that analyze evasive malware without being recognized) have also been studied extensively.

However, thus far, to the best of our knowledge, the following questions have not been answered with sufficient clarity.

– What proportion of malwares executes a certain type of anti-analysis operations? What types of anti-analysis operations do malwares execute frequently, and what types do they execute rarely?

– What virtual machine monitors or sandboxing systems do malwares attempt to detect?
– Do malwares execute a combination of multiple anti-analysis operations or only a single anti-analysis operation? What is the maximum number of anti-analysis operations executed by a single malware?
– Do real-world malwares that execute anti-analysis operations succeed in the detection of analysis systems?

We believe that researchers and engineers in this field need to gain deeper insights into the above-mentioned questions to effectively deal with recently developed malwares, which are becoming increasingly sophisticated.

The objective of this study is to clarify the recent trends of anti-analysis operations executed by real-world malwares. In this paper, we report on the results of analyzing the dynamic behavior log of 8243 samples of Windows malwares recorded in a malware analysis dataset, namely FFRI Dataset 2016 [3,4]. This dataset includes a complete log of Windows API calls invoked by all the malware processes.

In this paper, we present a case study that addresses the following points:

– The proportion of malwares that may become difficult to analyze if an analysis system does not adopt additional countermeasures against anti-analysis operations
– The types of anti-analysis operations against which security systems should preferentially adopt countermeasures
– The success ratio of each execution of anti-analysis operations
– The range of countermeasures needed by an analysis system to correctly analyze sophisticated malwares that execute a variety of anti-analysis operations
– The limitations of an existing dynamic analysis system that is widely used
– Insights that can be obtained only from dynamic malware analysis logs and insights that are difficult to obtain only from such logs

The remainder of this paper is organized as follows. Sect. 2 compares the present study with other related studies. Sect. 3 describes the anti-analysis operations targeted in this study. Sect. 4 presents an overview of the dataset used in this study. Sect. 5 presents the results of our analysis on malware behavior. Finally, Sect. 6 summarizes our findings and briefly explores directions for future studies.

2 Related Studies

Numerous studies have been conducted on the techniques by which a program determines the presence or absence of an analysis system. The studies by Garfinkel et al. [5] and Raffetseder et al. [6] provide rich surveys of existing techniques for the detection of virtual machine monitors and emulators. In these studies, the authors have only compared the techniques qualitatively; they did not clarify which techniques are adopted by actual malwares or what proportion of malwares execute operations based on each technique. The contribution of the present study is to present quantitative results that are useful for such clarifications.

Chen et al. [7] evaluated the differences between APT-attack malwares and general malwares in terms of the trends in operations that are aware of debuggers or virtual machines. Although their study was similar to the present study in that they analyzed the trends in anti-analysis operations using many malware samples, there are also many differences between the studies. First, in their study, anti-analysis operations were detected only through static analysis, while dynamic behavior was not considered. Moreover, they adopted a detection standard that is significantly less precise than that applied in the present study. For example, they regarded a call of the function `Sleep` as an anti-analysis operation. Furthermore, their study focused on the differences between two classes of malwares and did not clarify the details of the anti-analysis operations executed by malwares.

Kirat et al. [8] proposed *MalGene*, an automated technique for extracting the signatures of analysis evasion from system call sequences. Their technique is based on algorithms used in bioinformatics, data flow analysis, and data mining. In their evaluation of 2810 evasive malware samples, their system could automatically extract evasion signatures and classify them into 78 similar evasion techniques. It is interesting to apply their technique to API calls in the dataset we used in the present study. While their study aimed to develop techniques for analyzing system calls, the present study set out to uncover the detailed trends in the anti-analysis operations performed by recently developed real-world malwares.

Kirat et al. [9] proposed BareCloud, a system for automatically detecting evasive malware. They reported that their system could detect 5835 evasive malware instances out of 110,005 samples. Branco et al. [10] reported on the behavior related to anti-analysis operations using around four million malware samples. They showed that a considerable proportion of malwares executed these operations. Chubachi et al. [11] also studied anti-analysis operations using 89,119 real-world malware samples, and reported on the number of malware samples that attempted to detect each artifact of hypervisors or sandboxing systems. They also proposed SLIME, which is a system for analyzing analysis-aware malwares by monitoring their instruction execution and disguising resources checked in anti-analysis operations. In contrast to the present study, their studies did not present any extensive statistical data on the operations executed by the malwares.

Barbosa et al. [12] presented detailed statistical information on the anti-analysis characteristics of millions of malware samples. They leveraged static analysis, not dynamic analysis, and hence the class of characteristics they could find was substantially different from those found in the present study.

Ferrand [13] investigated the internal details of Cuckoo Sandbox and presented multiple methods by which it can be detected by malwares. As described in Sect. 4, FFRI Dataset was created with Cuckoo Sandbox. Ferrand also presented countermeasures to prevent detection and enable stealthier analysis with Cuckoo Sandbox. His work could influence trends in real-world malwares with regard to their awareness of Cuckoo Sandbox and facilitate the development of Cuckoo's cloaking mechanism against such malwares. His study and the present study are complementary. His study accelerated the development of anti-analysis techniques and stealthy monitoring techniques, whereas the present study provides a deeper understanding of the trends in the behavior of real-world malwares developed on the basis of these techniques.

Chen et al. [14] classified the behaviors of malware whereby they are aware of virtual machines or debuggers and evaluated the differences in the behavior of 6222 malware samples executed on physical and virtual machines managed by VMware Server. In contrast to the present study, they presented experimental results that included only the proportion of malwares that behave differently on a virtual machine. Moreover, their study did not concretely clarify the anti-analysis operations executed by the malwares.

Wang et al. [15] proposed a timing-based technique called hypervisor introspection, by which a program running a virtual machine can determine the presence of introspection operations executed by the hypervisor. Cardinal Pill Testing [16] is a method for detecting virtual machines through carefully designed tests with regard to CPU instruction semantics. nEther [17] is a method that detects the Ether Xen-based analysis system using both timing information and CPU semantics information. In contrast to their studies, the present study focused on the detection of virtualization mechanisms without the use of timing information or CPU instruction information, as described in Sect. 3.

In addition to the above-mentioned studies, many other studies have examined those methods for analyzing malwares that detect virtualization mechanisms and evade analysis [18–30]. These studies and the present study are complementary. We can further improve malware analysis methods by gaining a deeper understanding of the anti-analysis operations executed by malwares. Conversely, by adopting sophisticated analysis methods, we can understand malware behavior in greater detail.

3 Target Anti-Analysis Operations

Through anti-analysis operations, malwares attempt to find artifacts of analysis systems such as virtual machine monitors, sandboxing systems, and debuggers. Malwares can obtain a variety of information to perform anti-analysis operations [5, 6, 10, 13, 16, 17, 31, 32]. Ideally, all such information should be considered in this study. However, we concentrate on understanding anti-analysis operations with regard to the following information, because this information can be obtained from Windows API call sequences, which does not necessarily reveal other information:

(1) Computing resources: This refers to the presence, specification, and contents of computing resources, including files, folders, registry keys, and hardware information. Characteristic program files, configuration files, or folders are created in many guest operating systems running on a virtual machine to cooperate with the underlying virtual machine monitor. The specification of certain hardware devices, such as hard disks and network interface cards, also indicates a particular analysis system.
(2) Co-running programs: This refers to the names of daemon programs or applications that are simultaneously running in the same environment. A debugger process or other monitoring programs may be running simultaneously with the current process.
(3) Human interface: This refers to activities of human-interface devices such as a keyboard and mouse. There tends to be less human-computer interaction in an artificial environment created by an analysis system. Even if there is any such interaction, it is likely to be unnatural.

Malwares can attempt to obtain any of the above-mentioned types of information, any one of which is useful for detecting a virtualization mechanism.

Through anti-analysis operations, malwares can attempt to directly hinder analysis through various means including delaying their execution and removing hooks for function call interception. We also include these operations in the target.

This study does not deal with anti-analysis operations with regard to the following types of information, because it is difficult to understand them using FFRI Dataset:

(4) Values in memory and registers: This refers to a value stored in the start address of function code, values stored in special registers, values of particular flags in the process environment block, and so on. The start address of function code is often patched by debuggers to intercept a call. Values in debug registers and the value of the `BeingDebugged` flag in the block indicate whether the current process is being debugged.
(5) Results of special CPU instructions: This refers to the execution results of CPU instructions whose behavior in a virtual machine differs from that in a physical machine because of factors such as the omission of exact instruction emulation by the virtual machine monitor.
(6) Results of I/O operations: This refers to the execution results of instructions for operating I/O devices. The behavior of I/O devices in a virtual machine may differ from that in a real machine.
(7) Performance information: This refers to the time taken for the execution of a particular operation, patterns of assigning CPU time to processes, etc.

We exclude (4), (5), and (6) from the target because FFRI Dataset does not contain execution logs at the CPU instruction level. Further, it may be possible to recognize certain types of anti-analysis operations with regard to (7) because FFRI Dataset associates an invocation time with each Windows API call. However, we believe that the result will not be useful because many performance-based anti-analysis operations are expected be performed by executing CPU instructions, such as `rdtsc`, without invoking API calls, and FFRI Dataset does not contain any instruction-level log. Therefore, this study does not consider anti-analysis operations with regard to (7).

4 Dataset

FFRI Dataset [3, 4] is a dataset of dynamic analysis results obtained by executing malwares in Cuckoo Sandbox [33], which is a widely used open-source sandbox for malware analysis based on a virtual machine monitor. Four versions of FFRI Dataset are available (2013–2016); we choose the latest dataset, namely FFRI Dataset 2016. This study targets the analysis results of 8243 malware samples executed on Windows 10 (x64), which are contained in FFRI Dataset 2016. FFRI, Inc. collected these malware samples from January 2016 to March 2016 from all over the world using techniques such as Web crawling. Further, from these collected samples, 8243 samples judged as malware by more than 10 anti-virus products were selected. According to the dataset logs, Cuckoo Sandbox 2.0-dev and VirtualBox were used to create the dataset.

Table 1 Statistics information

Number of malware samples	8243
Number of child processes created by each malware sample (min, ave, max)	0, 0.68, 202
Number of API calls invoked by each process (min, ave, max)	0, 2721, 82337
Total number of API calls	37664465
Number of API functions used by each malware sample (min, ave, max)	0, 47.5, 140
Total number of API functions	288

Cuckoo Sandbox provides a mechanism to compare program behavior with the characteristics of malware behavior, which are referred to as *signatures*. Cuckoo Sandbox records the signatures detected in the analysis results. Functions for detecting signatures are maintained in a source tree called *community*, which is distinct from the source tree of the main part of Cuckoo Sandbox. FFRI Dataset 2016 contains signatures detected by the community code.

In addition, FFRI Dataset contains various types of information, including data extracted statically from malware binaries and communication logs created during execution. From this set of information, we extract (1) the signature detection results and (2) the call sequences of Windows API invoked by the malwares, and we analyze the combination of the two.

Table 1 lists the call sequence statistics information for FFRI Dataset 2016. In the table, min, ave, and max indicate the minimum, average, and maximum numbers, respectively. All the analysis results in the dataset contain logs of a process for which the program path is `C:\\Windows\\System32\\lsass.exe`. However, we excluded the logs from our analysis target because the process is not derived from a malware.

5 Analysis Results

We analyzed the dynamic analysis logs in FFRI Dataset 2016 using our programs, which transform the logs into a set of anti-analysis operation logs, and then summa-

Table 2 Top 20 signatures detected from the malwares

Rank	Description by Cuckoo Sandbox	Number and proportion of malware samples	Number and proportion of malware families
1	`File has been identified by at least one AntiVirus on VirusTotal as malicious`	8243 (100.0%)	3044 (100.0%)
2	`Performs some HTTP requests`	5332 (64.7%)	1976 (64.9%)
3	`Allocates read-write-execute memory (usually to unpack itself)`	4615 (56.0%)	1791 (58.8%)
4	`Generates some ICMP traffic`	4162 (50.5%)	1549 (50.9%)
5	`One or more potentially interesting buffers were extracted, these generally contain injected code, configuration data, etc.`	2858 (34.7%)	1077 (35.4%)
6	`Creates executable files on the filesystem`	2214 (26.9%)	748 (24.6%)
7	`One or more of the buffers contains an embedded PE file`	2116 (25.7%)	758 (24.9%)
8	`The executable has PE anomalies (could be a false positive)`	2006 (24.3%)	983 (32.3%)
9	`Executed a process and injected code into it, probably while unpacking`	1486 (18.0%)	624 (20.5%)
10	`Installs itself for autorun at Windows startup`	1378 (16.7%)	423 (13.9%)
11	`This executable has a PDB path`	1091 (13.2%)	387 (12.7%)
12	`Collects information to fingerprint the system (MachineGuid, DigitalProductId, SystemBiosDate)`	1029 (12.5%)	334 (11.0%)
13	`Code injection with CreateRemoteThread or NtQueueApcThread in a remote process`	579 (7.0%)	156 (5.1%)
14	`A process attempted to delay the analysis task.`	509 (6.2%)	95 (3.1%)
15	`Checks the version of Bios, possibly for anti-virtualization`	443 (5.4%)	100 (3.3%)
16	`This executable is signed`	432 (5.2%)	123 (4.0%)
17	`Detects VMWare through the presence of various files`	358 (4.3%)	55 (1.8%)
18	`Connects to a Dynamic DNS Domain`	310 (3.8%)	112 (3.7%)
19	`One or more processes crashed`	264 (3.2%)	116 (3.8%)
20	`Uses Windows utilities for basic Windows functionality`	204 (2.5%)	43 (1.4%)

rized them. In this section, we describe the results of this analysis.

5.1 Detected Signatures

We counted the number of malware samples and malware families from which the signatures were detected. We classified the malware samples into those malware families leveraging malware identification results by antivirus products included in the dataset. Although all the malware samples were examined by at least 36 antivirus products, we chose three major ones for the classification: those identified in the dataset as Microsoft, Avast, or Malwarebytes. We chose them because anti-malware market share reports by OPSWAT [34] in 2016 listed them as the top three products with the largest market shares. We labeled each malware with a concatenation of three malware family names assigned by the three products, and regarded the label as being the family name of the malware. Among the malware samples, 418 were not identified as malware by any of the three products, and so were excluded from the analysis. We then randomly chose one malware sample from each malware family and analyzed the logs of the samples. The number of resulting families was 3044. It should be noted that different signature sets can possibly be detected from different malware samples in one malware family, because their API calls do not necessarily agree completely.

Table 2 lists the top 20 signatures in descending order of the number of malware samples from which the signatures were determined. It also lists the number of malware families from which the top signatures were detected. The descriptions of the signatures are the original descriptions that appear in the analysis logs of Cuckoo Sandbox. The signatures of the anti-analysis operations (*anti-analysis signatures*, hereafter) are ranked 14, 15, and 17. Although the anti-analysis signatures are not highly ranked, we regard their numbers and proportions as being relatively significant.

The total number of signatures detected from all the malware samples was 41,626 (5.05 signatures per malware on average), while 90 signatures were detected from at least one malware sample.

Next, we counted the number of anti-analysis signatures detected. Cuckoo Sandbox stores the functions for detecting each signature in distinct files. In this study, we regard those files with the prefix `antidbg_`, `antiemu_`, `antisandbox_`, or `antivm_` as files that contain functions for detecting anti-analysis signatures. Hereafter, we refer to each signature by the name of the file storing the corresponding detection function.

Table 3 Numbers of anti-analysis signatures detected from the malwares

Signature name	Description by Cuckoo Sandbox	Number and proportion of malware samples	Number and proportion of malware families
`antisandbox_sleep`	`A process attempted to delay the analysis task.`	509 (6.2%)	95 (3.1%)
`antivm_generic_bios`	`Checks the version of Bios, possibly for anti-virtualization`	443 (5.4%)	100 (3.3%)
`antivm_vmware_files`	`Detects VMWare through the presence of various files`	358 (4.3%)	55 (1.8%)
`antivm_generic_scsi`	`Detects virtualization software with SCSI Disk Identifier trick(s)`	113 (1.4%)	68 (2.2%)
`antisandbox_forehwnd`	`Checks whether any human activity is being performed by constantly checking whether ...`	66 (0.80%)	20 (0.66%)
`antisandbox_idletime`	`Looks for the Windows Idle Time to determine the uptime`	37 (0.45%)	20 (0.66%)
`antidbg_windows`	`Checks for the presence of known windows from debuggers and forensic tools`	35 (0.42%)	16 (0.53%)
`antivm_vbox_keys`	`Detects VirtualBox through the presence of a registry key`	34 (0.41%)	24 (0.79%)
`antivm_vbox_files`	`Detects VirtualBox through the presence of a file`	29 (0.35%)	15 (0.49%)
`antivm_vmware_keys`	`Detects VMWare through the presence of a registry key`	14 (0.17%)	7 (0.23%)
`antivm_generic_disk`	`Queries information on disks, possibly for anti-virtualization`	11 (0.13%)	7 (0.23%)
`antisandbox_mouse_hook`	`Installs an hook procedure to monitor for mouse events`	7 (0.08%)	5 (0.16%)
`antivm_generic_services`	`Enumerates services, possibly for anti-virtualization`	7 (0.08%)	5 (0.16%)
`antivm_generic_firmware`	`Detects Virtual Machines through their custom firmware`	2 (0.02%)	1 (0.03%)
`antivm_sandboxie`	`Tries to detect Sandboxie`	2 (0.02%)	2 (0.07%)
`antiemu_wine`	`Detects the presence of Wine emulator`	1 (0.01%)	1 (0.03%)
`antisandbox_unhook`	`Tries to unhook Windows functions monitored by Cuckoo`	1 (0.01%)	0 (0.00%)
`antivm_virtualpc`	`Tries to detect VirtualPC`	1 (0.01%)	1 (0.03%)

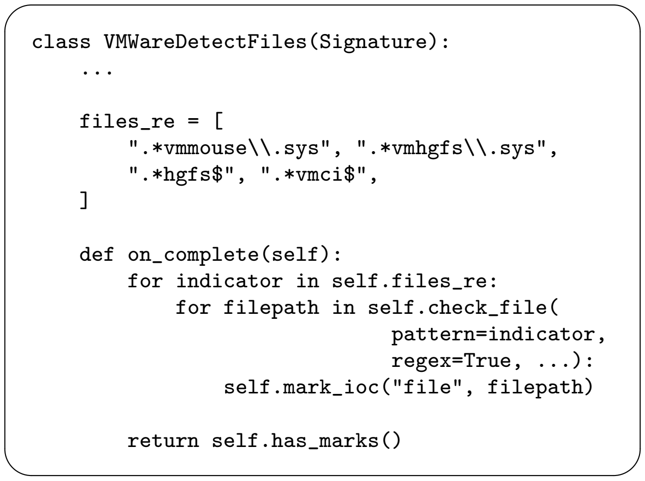
Table 3 lists the numbers of the anti-analysis signatures detected from the malware samples and malware families in descending order of the numbers of malware samples from which the signatures were detected. Among all the signatures of the virtualization mechanisms, the signatures of VMware and VirtualBox were detected from the largest and second-largest numbers of malware samples, respectively. The signatures of generic operations for detecting virtualization mechanisms were detected frequently, whereas the signatures of Sandboxie, VirtualPC, and Wine were rarely detected. Among the signatures of sandboxing systems, the signature of long sleeps and the signatures for monitoring foreground window and idle time were the most frequently detected signatures, whereas the signatures related to hook operations were rarely detected.

We observed nonnegligible changes in the proportions of malware samples, according to whether we used all the malware samples or only representative samples of the malware families. The proportion of some signatures such as `antisandbox_sleep` and `antivm_vmware_files` significantly decreased when using representative samples. This indicates that the dataset contained many variants of malware families from which such signatures were detected. On the other hand, the proportion of some signatures such as `antivm_generic_scsi` and `antivm_vbox_keys` significantly increased, indicating that the dataset did not contain so many variants of malware families from which such signatures were detected.

Table 4 lists the standards for anti-analysis signature detection. The descriptions of the standards are partly abbreviated and simplified owing to space limitations. For example, although the standard for `antivm_virtualpc` includes pattern matching of the accessed DLLs and files as well as mutexes, their description is omitted from the table. For full details on the standards, readers are encouraged to refer to the script programs under `modules/signatures/windows/` in the community source tree of Cuckoo Sandbox. Cuckoo Sandbox detects every anti-analysis signature using the information available in an API call sequence.

Table 4 Description of standards for signature detection (partly abbreviated and simplified)

Signature name	Detection standard
`antidbg_windows`	An argument labeled with `window_name` or `class_name` matches patterns including `OLLYDBG`
`antisandbox_forehwnd`	Both `GetForegroundWindow` and `NtDelayExecution` are invoked more than 100 times
`antisandbox_idletime`	`NtQuerySystemInformation` is invoked with argument `SystemProcessorPerformanceInformation`
`antisandbox_mouse_hook`	`SetWindowsHookExA` or `SetWindowsHookExW` is invoked with specified identifier values
`antisandbox_sleep`	The total amount of sleep time attempted is $\geq$ 120 s
`antisandbox_unhook`	Anomalies such as too many exceptions occur during analysis
`antivm_generic_bios`	An accessed registry key ends with `SystemBiosVersion` or `VideoBiosVersion`
`antivm_generic_disk`	`DeviceIoControl` is invoked with argument `IOCTL_DISK_GET_DRIVE_GEOMETRY`
`antivm_generic_scsi`	An accessed registry key matches patterns including ...`\\Services\\Disk\\Enum\\0`
`antivm_generic_firmware`	`NtQuerySystemInformation` is invoked with argument `SystemFirmwareTableInformation`
`antivm_generic_services`	`EnumServicesStatusA` or `EnumServicesStatusW` is invoked
`antivm_sandboxie`	An accessed file or loaded DLL matches ...`sbiedll(.dll)?`
`antivm_vbox_files`	An accessed file or loaded DLL matches patterns including `VBoxVideo.[a-zA-Z]{3}`
`antivm_vbox_keys`	An accessed registry key ends with `\\SOFTWARE\\Oracle\\VirtualBox Guest Additions`
`antivm_virtualpc`	A mutex whose name matches ...`MicrosoftVirtualPC7UserServiceMakeSure`... is opened
`antivm_vmware_files`	An accessed file matches patterns including ...`vmmouse.sys`, ...`vmhgfs.sys`, and ...`vmci`
`antivm_vmware_keys`	An accessed registry key matches patterns including ...`\\(Wow6432Node\\)?VMWare, Inc.`
`antiemu_wine`	An accessed registry key matches `\\HKEY_CURRENT_USER\\Software\\Wine`


```
class VMWareDetectFiles(Signature):
    ...

    files_re = [
        ".*vmmouse\\.sys", ".*vmhgfs\\.sys",
        ".*hgfs$", ".*vmci$",
    ]

    def on_complete(self):
        for indicator in self.files_re:
            for filepath in self.check_file(
                                pattern=indicator,
                                regex=True, ...):
                self.mark_ioc("file", filepath)

        return self.has_marks()
```

Fig. 1 Script for detecting an anti-analysis signature related to VMware (indents are slightly changed)

Figure 1 shows an example script for signature detection, which is extracted from the community source code. The script program, written in Python, detects the signature `antivm_vmware_files` by comparing file paths accessed during execution with four regular expressions that represent characteristic file paths usually employed in a guest operating system running on VMware. Cuckoo Sandbox records the accessed file paths and finally calls the function `on_complete` after the entire execution. This function performs pattern matching against each of the recorded paths and marks the analysis result with an indicator of compromise (IOC) if at least one file path matches with one of the regular expressions.

The community source code that is expected to be used to create the dataset includes 28 anti-analysis signatures (i.e., 28 signatures that are categorized by Cuckoo developers as `anti-debug`, `anti-emulation`, `anti-sandbox`, or `anti-vm`). Although the following 10 signatures are provided in the community code in addition to the 18 signatures listed in Table 3, they were not detected from any malware samples in the dataset:

`antidbg_devices`, `antisandbox_file`, `antisandbox_sunbelt`, `antivm_generic_ide`, `antivm_vbox_acpi`, `antivm_vbox_devices`, `antivm_vbox_window`, `antivm_virtualpc_magic`, `antivm_vmware_in_insn`, `shutdown`

Although the community source code as of August 12, 2016, additionally provides the following seven signatures, none of them was detected as might be expected:

`antivm_bochs_keys`, `antivm_computername`, `antivm_generic_cpu`, `antivm_hyperv_keys`, `antivm_parallels_keys`, `antivm_vpc_keys`, `antivm_xen_keys`

Because these seven signatures are not expected to have been introduced into the environment for dataset creation, Cuckoo Sandbox did not detect any signature related to Bochs, Hyper-V, Parallels, or Xen. Hence, we attempted to find these new signatures by applying their detection standard to the dataset. Nevertheless, none of these signatures was detected. However, we found that remarkable strings, such as `Hyper-V` and `microsoft-hyper-v`, appeared in the arguments of many API calls. Thus, we did not exclude the possibility that some of the malware samples attempt to recognize Hyper-V.

Next, we examined how many signatures were detected from each malware sample and each malware family. Table 5 shows the results obtained when counting all the signatures, not only the anti-analysis ones.

Table 5 Number of signatures detected from each malware sample or each malware family

Number of signatures	Number and proportion of malware samples	Number and proportion of malware families
0	0 (0.00%)	0 (0.00%)
1	724 (8.8%)	241 (7.9%)
2	923 (11.2%)	322 (10.6%)
3	1548 (18.8%)	540 (17.7%)
4	1103 (13.4%)	399 (13.1%)
5	880 (10.7%)	369 (12.1%)
6	848 (10.3%)	348 (11.4%)
7	570 (6.9%)	242 (8.0%)
8	520 (6.3%)	235 (7.7%)
9	348 (4.2%)	126 (4.1%)
10	213 (2.6%)	74 (2.4%)
11	159 (1.9%)	57 (1.9%)
12	158 (1.9%)	41 (1.3%)
13	97 (1.2%)	20 (0.66%)
14	96 (1.2%)	19 (0.62%)
15	47 (0.60%)	9 (0.30%)
16	8 (0.10%)	2 (0.07%)
17	1 (0.01%)	0 (0.00%)
≥ 18	0 (0.00%)	0 (0.00%)

The maximum and minimum number of signatures detected from one malware sample was 17 and 1, respectively. This result indicates that some "active" malwares conduct a large block of operations that are determined as signatures. The malware sample from which 17 signatures were detected will be elaborated as Malware 1 in Sect. 5.6. The maximum number of signatures detected from one representative sample of a malware family was 16, not 17, because the above-mentioned malware sample (Malware 1) was not chosen as a representative from the corresponding family. We did not find a large difference in the distribution of the number of signatures between when using all the malware samples and when using samples that are representative of a family.

Table 6 lists the results obtained when counting anti-analysis signatures only. In the same way as with the previous results, we did not find a large difference in the distribution between when using all the malware samples and when using samples that are representative of a family. At least one anti-analysis signature was detected from the execution of around 10.4% of the malware samples. In general, only a few anti-analysis signatures were detected from one malware sample. More than three anti-analysis signatures were detected from only around 0.15% of all the malware samples. Further, more than four anti-analysis signatures were detected from only five out of the 8243 samples. This result indicates that only a small proportion of malwares execute a wide range of anti-analysis operations.

Table 6 Number of anti-analysis signatures detected from each malware sample or each malware family

Number of anti-analysis signatures	Number and proportion of malware samples	Number and proportion of malware families
0	7387 (89.6%)	2791 (91.7%)
1	416 (5.0%)	151 (5.0%)
2	83 (1.0%)	26 (0.85%)
3	345 (4.2%)	69 (2.3%)
4	7 (0.08%)	3 (0.10%)
5	5 (0.06%)	4 (0.13%)
≥ 6	0 (0.00%)	0 (0.00%)

We examined each of the five malware samples from which five anti-analysis signatures were detected. The signatures `antivm_generic_bios`, `antivm_generic_scsi`, `antivm_vbox_keys`, and `antivm_vmware_keys` were detected from all these samples. As the remaining signature, `antidbg_windows` was detected from two malware samples and `antisandbox_sleep` was detected from three malware samples.

5.2 Programs that Execute Anti-Analysis Operations

Some malwares create a new process during their execution. Anti-analysis operations may be executed by the first process of a malware or by a child process created during the execution.

Some child processes can execute a downloaded malicious program, while others can simply execute a benign program to help the main process of the malware. In addition, some child processes can first execute a benign program and then become targets of shellcode injection, which is typically achieved with the function `CreateRemoteThread`. As shown in Table 2, the signature of code injection into a remote process was detected from 579 malware samples.

We examined the names of the programs from which an anti-analysis signature was detected, and counted the number of API calls for anti-analysis operations invoked by each of the programs (including both successful and unsuccessful calls). We also counted the number of processes from which an anti-analysis signature was detected.

We excluded the signatures `antisandbox_forehwnd` and `antisandbox_sleep` from this result because the numbers of API calls for these anti-analysis operations are not meaningful.

Tables 7 and 8 show the results. Each malware file in FFRI Dataset 2016 is named as "the SHA-1 hash value of the file content"`.exe`. The tables show only high-ranked programs in terms of the total number of API calls. After these programs, the ranking continues

Table 7 Number of API calls invoked for anti-analysis operations

Program name	Sum	`antivm_generic_bios`	`antivm_debug_windows`	`antivm_vmware_files`	`antivm_generic_scsi`	`anti sandbox_idletime`	`antivm_vbox_keys`	`antivm_vbox_files`	`antivm_vmware_keys`	Others
Hashvalue`.exe`	3019	436	1375	17	457	526	86	48	20	54
`explorer.exe`	1916	1104		708	38	1	15	27	11	12
`hh.exe`	189	84			69		21		15	
`helppane.exe`	144	64			50		16		14	
`regedit.exe`	127	56	2		41		14		14	
`bfsvc.exe`	62	28			26		7		1	
`powershell.exe`	62					62				
`notepad.exe`	57	24			24	3	6			
`write.exe`	54	24			24		6			
`iesecure.exe`	52			26				26		

Table 8 Number of processes from which an anti-analysis signature was detected

Program name	Sum	`antivm_generic_bios`	`antivm_debug_windows`	`antivm_vmware_files`	`antivm_generic_scsi`	`anti sandbox_idletime`	`antivm_vbox_keys`	`antivm_vbox_files`	`antivm_vmware_keys`	Others
Hashvalue`.exe`	383	84	40	13	110	31	34	29	15	27
`explorer.exe`	734	357		354	3	1	3	9	1	6
`hh.exe`	13	4			4		4		1	
`helppane.exe`	7	2			2		2		1	
`regedit.exe`	9	2	1		2		2		2	
`bfsvc.exe`	13	4			4		4		1	
`powershell.exe`	5					5				
`notepad.exe`	10	3			3	1	3			
`write.exe`	9	3			3		3			
`iesecure.exe`	26			13				13		

with `splwow64.exe`, `winhlp32.exe`, `eventvwr.exe`, `verifiergui.exe`, `1.exe`, `MicrosoftTray.exe`, `M.exe`, `Sdat.exe`, `kernel21.exe`, and `diskchk.exe`. All these programs invoked less than 20 API calls for anti-analysis operations. Moreover, the tables show only high-ranked signatures in terms of the total numbers of API calls.

The result indicates that anti-analysis operations are often executed by the first process of a malware as well as by other processes. All programs except hash-value programs in the tables are utilities provided by the operating systems. These utilities could invoke anti-analysis operations for benign purposes, or shellcode injected into these utilities could invoke anti-analysis operations for malicious purposes.

Signatures of generic anti-analysis operations and signatures related to the registry keys of virtual machine monitors were detected from a wide range of programs. The other signatures were detected from only a small number of programs. A strong correlation was observed between `powershell.exe` and `antisandbox_idletime` as well as among `explorer.exe`, `iesecure.exe`, and signatures related to the files of virtual machine monitors.

5.3 Discussion on Individual Signatures

5.3.1 Overview

To determine whether the malwares succeeded in anti-analysis operations, we examined the number of successful and unsuccessful API calls that match each anti-analysis signature. We counted the number by comparing the standards used by Cuckoo Sandbox for signature detection with the function names and argument values of all API calls invoked by malwares from which an anti-analysis signature was detected. Cuckoo Sandbox ignores some unsuccessful API calls that match an anti-analysis signature. However, in this study, we count and report the numbers of both successful and unsuccessful calls because the purpose of this study is to understand as much anti-analysis behavior of malwares as possible. We determined the success and failure of each API call based on the return value, the `NTSTATUS` error code, and the specification of the API function. Although some API functions are exceptional, a call is determined to be a failure if the return value or `NTSTATUS` is negative, and a success otherwise.

Table 9 Success and failure of API calls for anti-analysis operations

Signature name	Number of successful calls	Number of unsuccessful calls
antisandbox_sleep	—	—
antivm_generic_bios	1503	345
antivm_vmware_files	0	751
antivm_generic_scsi	745	0
antisandbox_forehwnd	—	—
antisandbox_idletime	587	5
antidbg_windows	0	1379
antivm_vbox_keys	0	174
antivm_vbox_files	0	101
antivm_vmware_keys	0	76
antivm_generic_disk	25	0
antisandbox_mouse_hook	6	2
antivm_generic_services	7	7
antivm_generic_firmware	10	6
antivm_sandboxie	0	3
antiemu_wine	0	2
antisandbox_unhook	—	—
antivm_virtualpc	0	1

Table 9 shows the result. The numbers of some anti-analysis operations are not filled in the table because the numbers do not indicate the number of these anti-analysis operations. All calls of anti-analysis operations related to virtualization mechanisms (i.e., `antivm_vmware_*`, `antivm_vbox_*`, `antivm_sandboxie`, `antiemu_wine`, and `antivm_virtualpc`) failed. On the other hand, all calls of generic anti-analysis operations that check disk information (`antivm_generic_scsi` and `antivm_generic_disk`) were successful. The calls of generic anti-analysis operations that check the BIOS version (`antivm_generic_bios`) included successful and unsuccessful ones. The calls of generic anti-analysis operations that check the firmware information and enumerate the services (`antivm_generic_firmware` and `antivm_generic_services`) were all successful practically, as described later in detail. In the following subsections, we explain the details of malware behavior related to each anti-analysis signature.

5.3.2 Sleeps

Some malwares sleep to delay analysis; thus, Cuckoo Sandbox regards a large amount of accumulative sleep time as a signature. It calculates the sum of the times given as arguments of `NtDelayExecution` and compares the sum with a threshold.

This signature was detected from the largest number of malware samples among all the signatures. For this signature, the number of related API calls is not meaningful because only the total amount of sleep time matters.

A long sleep is easy to implement and effectively hinders analysis. Hence, this signature was naturally detected from many malware samples. However, it is important to note that not all long sleeps are necessarily intended for anti-analysis operations; some malwares might sleep for other purposes, such as waiting for communication.

5.3.3 BIOS Version Checks

Many virtual machine monitors provide an inherent virtual BIOS. The BIOS version information enables malwares to determine the presence and identify the name of an underlying virtual machine monitor, if any, with high accuracy.

In general, the BIOS version information is used for various purposes besides anti-analysis operations. Hence, BIOS version checks executed by a program do not immediately imply that the program attempts to detect artifacts of a virtual machine monitor. Actually, Cuckoo Sandbox describes the signature with a subtle expression, `possibly for anti-virtualization`. Similar expressions are used to describe signatures of `antivm_generic_disk` and `antivm_generic_services`. API calls that match these signatures are not necessarily for anti-analysis operations.

The signature of BIOS version checks was detected from 443 malware samples. Success or failure of the calls was dependent solely on an accessed registry key. Accesses to keys located under `HKEY_LOCAL_MACHINE\\HARDWARE` were successful, whereas accesses to keys located under `HKEY_LOCAL_MACHINE\\SOFTWARE` failed. Successful calls received the string `LENOVO - 2020` or `LENOVO - 3220` for the registry key `SystemBiosVersion`, and they received the string `Hardware Versio` for the registry key `VideoBiosVersion`.

Cuckoo Sandbox returns spurious BIOS information that differs from the information on the physical BIOS. For `SystemBiosVersion`, it randomly chooses one of the two above-mentioned strings and returns it. For `VideoBiosVersion`, it returns the string `Hardware Version 0.0`. Because the strings are publicly available, it is not difficult to write a program that determines whether it is running in Cuckoo Sandbox. It is possible for some malware samples attempting to recognize Cuckoo Sandbox with this operation to correctly determine the presence of Cuckoo Sandbox.

5.3.4 Detection of Virtual Machine Monitors

The malwares in the dataset failed in all API calls for detecting virtual machine monitors. Conversely, they were successful in most attempts to correctly predict the *absence* of virtual machine monitors. The malwares naturally failed in opening files or registry keys related to VMware because the running environment was built with VirtualBox. A signature of operations for recognizing VirtualPC was detected only once from a call to the function `NtCreateMutant` with the argument `MicrosoftVirtualPC7UserServiceMakeSureWe'reTheOnlyOneMutex`. This call also failed, and the failure is natural for the same reason as that stated above. All attempts to open files or registry keys related to VirtualBox also failed because resources related to VirtualBox Guest Additions were not found in the guest OS. Cuckoo Sandbox uses accesses to files of VirtualBox Guest Additions as a detection standard of signatures related to VirtualBox; all such file accesses failed in the execution recorded in the dataset.

5.3.5 Detection of Sandboxie and Wine

Sandboxie is not a virtual machine monitor but software that creates a virtual environment in the operating system for running untrusted programs. Operations onto resources, such as files, performed in a virtual environment are applied to virtualized counterparts and do not affect the real resources. Signatures related to Sandboxie were detected from two malware samples. One of them invokes the function `LdrLoadDll` twice with the Sandboxie library `SbieDll.dll` as an argument. The other invokes the function `NtCreateMutant` only once with `Sandboxie_SingleInstanceMutex_Control` as an argument. All the calls failed.

The signature related to Wine, which is a Windows emulator, was detected from one malware sample. The malware accessed the registry key `HKEY_CURRENT_USER\\Software\\WINE` twice with the function `RegOpenKeyExW`, and it failed in both accesses.

The malwares naturally failed in all API calls related to the Sandboxie and Wine signatures because it is highly unlikely that either Sandboxie or Wine was installed in the environment.

5.3.6 Checks of Disk Hardware Information

Many virtual machine monitors provide inherent virtual disk hardware to virtual machines. Disk hardware information is a critical clue for judging whether the running environment is a virtual one. Cuckoo Sandbox detects the signature `antivm_generic_scsi` from API calls for accessing registry keys of SCSI device identifiers or disk services. In general, these keys maintain data for the identification of disk hardware, such as product numbers.

Cuckoo Sandbox provides a mechanism called "VM cloaking", in which spurious hardware and software information is provided to virtual machines to deceive malware. Specifically, when the original value of a SCSI device registry contains a predefined substring such as `vbox`, `vmware`, `qemu`, or `virtual`, Cuckoo Sandbox replaces the value with a random character string or with the string `ST9160411AS`, which represents a Seagate hard disk. The string `ST9160411AS` appeared in the argument strings of 33 malware samples in the dataset.

The signature `antivm_generic_disk` was detected from various API calls, which include `NtCreateFile` calls with file paths containing the substring `physicaldrive0` or `scsi0`. They also include `DeviceIoControl` calls with special control code. All API calls for operations related to these signatures were successful. If some malware samples in the dataset determined the presence of virtualization mechanisms using disk hardware information, they might have correctly judged that they were running in a virtual environment. As described above, some malware samples actually obtained publicly available information that Cuckoo Sandbox provides to disguise environments.

5.3.7 Checks of Human Activity

Some malwares attempt to detect the characteristic behavior related to human activity in order to identify whether the running environment is a sandbox for analysis. Cuckoo Sandbox supports the signatures of such behavior, namely `antisandbox_forehwnd`, `antisandbox_idletime`, and `antisandbox_mouse_hook`.

For `antisandbox_forehwnd`, Cuckoo Sandbox makes a decision based on the number of calls by malwares to `GetForegroundWindow`, which is a function for obtaining the handle of the foreground window, and `NtDelayExecution`, which is a function for sleeping. In general, changes in the foreground window indicate human activity. Many malware samples in the dataset repeatedly invoked `GetForegroundWindow` and `NtDelayExecution` numerous times. We counted the maximum and average numbers of `GetForegroundWindow` calls and `NtDelayExecution` calls invoked by malwares from which the signature `antisandbox_forehwnd` was detected. The maximum and average numbers of `GetForegroundWindow` calls were 9960 and 1040, respectively, while the maximum and average numbers of `NtDelayExecution` calls were 5172 and 1067, respectively.

For `antisandbox_idletime`, Cuckoo Sandbox looks for API calls to determine the amount of time for which the system has been idle. In general, a long idle time indicates a long uptime, and a long uptime indicates an ordinary environment not intended for malware analysis. Most API calls for operations related to `antisandbox_idletime` were successful, and only five calls were unsuccessful. All the unsuccessful calls were caused by insufficient buffer sizes. Immediately after each of the unsuccessful calls, the malwares invoked the same API function again and succeeded in it.

For `antisandbox_mouse_hook`, Cuckoo Sandbox looks for attempts by the malwares to intercept mouse events. Six calls for operations related to this signature were successful and two were unsuccessful. The unsuccessful calls were caused by invalid code addresses given to arguments. It is not straightforward to determine whether the malware samples that executed the successful calls recognized the analysis environment, because Cuckoo Sandbox automatically moves the mouse cursor and clicks mouse buttons in a virtual machine to disguise the virtual machine as an ordinary environment.

The signature `antisandbox_forehwnd` was detected frequently compared to the other anti-analysis signatures. Further, `antisandbox_idletime` was also detected relatively frequently. On the other hand, `antisandbox_mouse_hook` was detected from only a few malware samples. We surmise from this result that malware developers prefer obscure anti-analysis operations, such as simply reading system information, rather than conspicuous operations, such as inserting hooks into API calls.

5.3.8 Checks of Debuggers and Forensic Tools

Some malwares check for the presence of windows from debuggers and forensic tools in the running environment. Cuckoo Sandbox detects the signature `antidbg_windows` from API calls whose argument labeled with `window_name` or `class_name` matches strings that characterize debuggers and forensic tools. The strings include `OLLYDBG`, `WinDbgFrameClass`, `FilemonClass`, and `Registry Monitor - Sysinternals: www.sysinternals.com`.

The malware samples in the dataset invoked 1379 API calls for operations related to this signature. All of them were calls of either `FindWindowA` or `FindWindowW`, which are functions for finding a window with a class name and window name that matches given argument strings. All 1379 calls failed because none of the searched windows existed in the environment. Hence, the malwares were likely to judge that no debugger or forensic tool was running, which is consistent with the facts.

A well-known method for a malware to check whether it is being debugged is calling the function `IsDebuggerPresent`. However, Cuckoo Sandbox does not have any signature detection standard based on the call of this function. Although many malware samples in the dataset called `IsDebuggerPresent`, Cuckoo Sandbox simply ignored the calls. The number of malwares that called the function at least once was 2900, and the total number of calls was 4559, out of which 4555 failed and only four were successful. All the successful calls were invoked by the second process that was dynamically created by the first malware process. The memory of the second process was read and written by the first process with the functions `ReadProcessMemory` and `WriteProcessMemory`, respectively.

5.3.9 Enumeration of Services

The signature `antivm_generic_services` is detected from calls of API functions whose prefix is `EnumServiceStatus`. These functions enumerate the currently running services. The presence of particular services indicates the presence of analysis software.

This signature was detected from only two malware samples, and only 14 API calls were invoked for operations related to this signature. Among the 14 calls, seven calls were successful and the other seven calls were unsuccessful. However, the unsuccessful calls were only for returning the error code `ERROR_MORE_DATA`, which represents the continuation of data. Hence, practically, all attempts by the malwares for the operations were successful. It is considered that the malwares successfully recognized the services running in the operating system.

5.3.10 Checks of Firmware Information

The signature for checking firmware information, namely `antivm_generic_firmware`, was detected from only two malware samples, and only 16 API calls were invoked for operations related to the signature. Both malware samples called the function `NtQuerySystemInformation` with the argument `SystemFirmwareTableInformation`, and the signature was detected from these calls. Among the 16 calls, 10 calls were successful and six calls were unsuccessful. All the unsuccessful calls were caused by insufficient buffer sizes for storing the results. Immediately after the unsuccessful calls, the malwares called the same function again, and these second attempts were all successful.

Hence, the malwares successfully obtained the firmware information in all attempts of the operations.

5.4 Whether the Malwares Recognized Virtualization

We consider that only a few malware samples from the dataset recognized Cuckoo Sandbox because many malware samples did not terminate themselves and continued their execution long after the anti-analysis operations. We examined the position of the last successful API call for operations related to each anti-analysis signature in the API call sequence of each malware process. Table 10 shows the average positions. Here, the *position* of a call in a sequence indicates the order of the call (smaller is earlier) divided by the number of calls contained in the sequence. In addition, the table also shows the average number of subsequent API calls invoked after the last successful call until process termination. If a malware was programmed to terminate its execution immediately after the successful detection of an analysis system, API calls for anti-analysis operations would be observed in the final stage of the call sequence. However, API calls for anti-analysis operations were not actually observed in such stage. Moreover, the malwares executed more API calls than the expected number required in a termination operation (i.e., ∼100 calls). From the viewpoint of not only the relative positions but also the absolute numbers of calls, it is reasonable to assume that a majority of the malwares did not successfully detect Cuckoo Sandbox and terminate execution.

We also conducted a microscopic examination of whether the malwares recognized Cuckoo Sandbox. We investigated the behavior observed after the last successful API call described above. Because some sophisticated malwares execute dummy or misleading operations after they had detected an analysis system [35], the behavior of API calls, as well as their numbers and positions, should be considered. The targets of the investigation were malware samples from which the signature `antisandbox_mouse_hook`, `antivm_generic_disk`, `antivm_generic_firmware`, or `antivm_generic_services` was detected. We chose these targets because of the small numbers of the samples. We found that 24 out of 28 malwares attempted at least one of the following operations: file write, registry key write, process creation, network communication to the external network, and thread creation in another process. These operations are mainly used for malicious purposes. These 24 malware samples are expected to continue malicious operations because they did not or could not detect Cuckoo Sandbox. The remaining four malware samples terminated themselves without executing operations that can be regarded as malicious.

Table 10 Average relative positions of the last successful API call related to an anti-analysis operation and average numbers of API calls invoked after the last API call

Signature name	Position of last call	Number of subsequent calls
`antivm_generic_bios`	21.6%	2865
`antivm_generic_scsi`	69.2%	1353
`antisandbox_idletime`	84.2%	1049
`antivm_generic_disk`	37.7%	8125
`antisandbox_mouse_hook`	31.0%	301
`antivm_generic_firmware`	44.7%	524
`antivm_generic_services`	19.6%	300

5.5 Overlooked Behavior

No security software can possibly detect all types of anti-analysis operations. In fact, FFRI Dataset 2016 contains numerous signs of anti-analysis operations that seem to be overlooked by Cuckoo Sandbox. Here, we briefly describe several reasons for such oversight that we consider important.

– Arguments of some API functions are not checked even though they should be. For example, Cuckoo Sandbox does not scan the arguments of the function `LdrGetDllHandle`; consequently, it ignored several characteristic strings that appeared in the arguments of `LdrGetDllHandle` in FFRI Dataset 2016. Examples include `sbiedll.dll` (a library file of Sandboxie) and `api_log.dll` (a library file of SunBelt Sandbox).
– Pattern matching of files and registry keys is not exhaustive. For example, although some malwares attempted to open the file `vmmemctl`, which is a file used by VMware, Cuckoo Sandbox ignored them owing to the lack of a pattern for signature detection that matches the file name.
– Arguments of unsuccessful calls of some API functions are not checked. For example, the arguments of the function `NtCreateFile` are excluded from the target of pattern matching if the call fails. The arguments of unsuccessful calls are simply ignored. As a result, Cuckoo Sandbox ignored many operations that were considered highly likely to be anti-analysis operations. Further, it ignored all arguments containing a string that characterizes a virtual machine monitor, such as `VBoxGuest`. We consider that invocation attempts of API calls matching the patterns themselves should be detected as signatures, regardless of whether they are successful, because

analysis systems should enable malware analysts to keep track of all attempts of anti-analysis operations exhaustively. Providing different signatures to successful anti-analysis operations and unsuccessful ones may further facilitate malware analysis.

5.6 Behavior of Individual Malwares

The following paragraphs describe the behavior of several remarkable malware samples.

Malware 1 (Hash Value 8a127994...): The number of signatures detected from this malware sample (17) was the largest among all the malware samples. The detected signatures include four anti-analysis signatures, namely `antisandbox_idletime`, `antisandbox_sleep`, `antivm_generic_bios`, and `antivm_vmware_files`. After the anti-analysis operations, this malware sample attempted to execute a wide range of operations including network communication and did not show the behavior of immediate termination.

Malware 2 (Hash Value bef62f27...): The behavior of this malware sample is rare in the sense that this is one of two malware samples from which the signature related to Sandboxie was detected, one of two malware samples from which the signature related to firmware information checks was detected, and the only malware sample from which the signature related to Wine was detected. This malware sample attempted to open several characteristic files such as `VBoxGuest`, `HGFS`, and `vmci` with the function `NtCreateFile`. We regard these as being highly likely to be anti-analysis operations for detecting VirtualBox or VMware. However, Cuckoo Sandbox ignored the open operations for the reason described in Sect. 5.5. In addition, this malware sample executed many apparently anti-analysis operations. For example, it checked the free disk space, the number of processors, and the location of the mouse cursor. It invoked the API function `GetCursorPos` as many as 5816 times.

Malware 3 (Hash Value f6772412...): Rare signatures including firmware information checks, Sandboxie detection, and VirtualPC detection were detected from this malware sample. This malware sample immediately began to terminate itself after the execution of these anti-analysis operations. Here, it should be noted that this malware failed to detect Sandboxie or VirtualPC. Hence, if this malware terminated execution because of being analyzed, the detection of the analysis was likely to be caused by another information source, such as firmware or performance. Thus far, it has not been determined whether the termination is because of the detection of Cuckoo Sandbox. Therefore, further investigation based on other information, such as malware bodies and executed instruction sequences, is required. This malware sample also executed other interesting operations, although they were not reported as signatures. For example, it obtained the number of CPUs with the function `GetNativeSystemInfo`, obtained the cursor position periodically every 100 ms, and attempted to open a folder named `C:\\Sandbox`.

6 Summary and Future Work

We reported the trends of anti-analysis operations executed by recently developed malwares whose dynamic behavior was recorded in FFRI Dataset 2016. Our findings can be summarized as follows:

– Among 8243 malware samples, 856 (10.4%) samples executed at least one type of the 28 anti-analysis operations investigated in this study. Most of them executed only one, two, or three types of anti-analysis operations. No malware sample executed more than five types of anti-analysis operations.
– VMware was the virtual machine monitor that the largest number of malware samples was aware of. VirtualBox was in the next position. Behavior related to Bochs, Hyper-V, Parallels, and Xen was not detected probably because the signatures of the behavior had not been incorporated in Cuckoo Sandbox when the dataset was created. We found that the dataset contains many API calls whose argument strings are related to Hyper-V.
– Anti-analysis operations were often executed by the first process of a malware by itself. They were also executed with nonnegligible frequency by child processes created by the malware. Further, `explorer.exe`, `hh.exe`, and `helppane.exe` were the top three Windows programs running in child processes that executed the largest number of anti-analysis operations.
– We surmise that the malwares failed in all attempts to detect a virtual machine monitor by accessing individual files or registry keys. Conversely, they were successful in most attempts to correctly predict the *absence* of virtual machine monitors. On the other hand, thus far, it has not been determined whether the malwares could successfully detect virtual machine monitors using hardware information or service information.
– Some malware samples obtained hardware information and then terminated themselves without exe-

cuting any operation with the original purpose of the malware.
– Cuckoo Sandbox ignored some operations that were likely to be anti-analysis operations. One reason for this oversight is that the patterns of functions and arguments for signature detection are not exhaustive. Another reason is that Cuckoo Sandbox ignores arguments of some unsuccessful calls.

These are the trends observed only in FFRI Dataset 2016, which was collected with Cuckoo Sandbox 2.0-dev. Further investigation is needed to determine whether these trends are universal ones that are also observed when analyzing a general set of recently developed malwares and when using other analysis systems.

There are several directions for future work. First, future studies are expected to clarify malware behavior that cannot be understood with only API call sequences. An extremely important example of the behavior is time-based detection of analysis systems. It would be necessary to combine the insights of the present study with those obtained by analyzing performance data or a trace of executed CPU instructions. Second, it would be necessary to develop a method capable of more accurately predicting whether and when malwares successfully detect the presence of analysis systems. Although some malware samples in the dataset finished their execution without performing any malicious operations, the reasons for their behavior have not been fully identified. Moreover, sophisticated malwares in the dataset might possibly execute dummy or misleading operations after the detection of analysis systems [35]. One starting point for future work will be to gain a full understanding of the behavior observed around the last execution of anti-analysis operations.

Acknowledgements We are grateful to Yosuke Chubachi for useful suggestions. 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 Number 26330080.

References

1. VENOM vulnerability. CVE-2015-3456 (2015)
2. Symantec: Internet security threat report, Volume 21, April 2016 (2016)
3. Hatada, M., Akiyama, M., Matsuki, T., Kasama, T.: Empowering anti-malware research in Japan by sharing the MWS datasets. Journal of Information Processing **23**(5), 579–588 (2015)
4. Takata, Y., Terada, M., Murakami, J., Kasama, T., Yoshioka, K., Hatada, M.: Datasets for anti-malware research ∼MWS datasets 2016∼. In: IPSJ SIG Technical Report, vol. 2016-CSEC-74 (2016)
5. Garfinkel, T., Adams, K., Warfield, A., Franklin, J.: Compatibility is not transparency: VMM detection myths and realities. In: Proceedings of the 11th Workshop on Hot Topics in Operating Systems (2007)
6. Raffetseder, T., Kruegel, C., Kirda, E.: Detecting system emulators. In: Proceedings of the 10th Information Security Conference, pp. 1–18 (2007)
7. Chen, P., Huygens, C., Desmet, L., Joosen, W.: Advanced or not? A comparative study of the use of anti-debugging and anti-VM techniques in generic and targeted malware. In: Proceedings of the 31st IFIP International Conference on ICT Systems Security and Privacy Protection, pp. 323–336 (2016)
8. Kirat, D., Vigna, G.: MalGene: Automatic extraction of malware analysis evasion signature. In: Proceedings of the 22nd ACM Conference on Computer and Communications Security, pp. 769–780 (2015)
9. Kirat, D., Vigna, G., Kruegel, C.: BareCloud: Bare-metal analysis-based evasive malware detection. In: Proceedings of the 23rd USENIX Security Symposium, pp. 287–301 (2014)
10. Branco, R.R., Barbosa, G.N., Neto, P.D.: Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-VM technologies. Black Hat USA 2012 (2012)
11. Chubachi, Y., Aiko, K.: SLIME: Automated anti-sandboxing disarmament system. Black Hat Asia 2015 (2015)
12. Barbosa, G.N., Branco, R.R.: Prevalent characteristics in modern malware. Black Hat USA 2014 (2014)
13. Ferrand, O.: How to detect the Cuckoo Sandbox and to strengthen it? Journal of Computer Virology and Hacking Techniques **11**(1), 51–58 (2015)
14. Chen, X., Andersen, J., Mao, Z.M., Bailey, M., Nazario, J.: Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In: Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, pp. 177–186 (2008)
15. Wang, G., Estrada, Z.J., Pham, C., Kalbarczyk, Z., Iyer, R.K.: Hypervisor introspection: A technique for evading passive virtual machine monitoring. In: Proceedings of the 9th USENIX Workshop on Offensive Technologies (2015)
16. Shi, H., Alwabel, A., Mirkovic, J.: Cardinal pill testing of system virtual machines. In: Proceedings of the 23rd USENIX Security Symposium, pp. 271–285 (2014)
17. Pék, G., Bencsáth, B., Buttyán, L.: nEther: In-guest detection of out-of-the-guest malware analyzers. In: Proceedings of the 4th European Workshop on System Security (2011)
18. Balzarotti, D., Cova, M., Karlberger, C., Kruegel, C., Kirda, E., Vigna, G.: Efficient detection of split personalities in malware. In: Proceedings of the 17th Annual Network and Distributed System Security Symposium (2010)
19. Blackthorne, J., Bulazel, A., Fasano, A., Biernat, P., Yener, B.: AVLeak: Fingerprinting antivirus emulators through black-box testing. In: Proceedings of the 10th USENIX Workshop on Offensive Technologies (2016)
20. Dinaburg, A., Royal, P., Sharif, M., Lee, W.: Ether: Malware analysis via hardware virtualization extensions. In: Proceedings of the 15th ACM Conference on Computer and Communications Security, pp. 51–62 (2008)
21. Kang, M.G., Yin, H., Hanna, S., McCamant, S., Song, D.: Emulating emulation-resistant malware. In: Proceedings of the 2nd ACM Workshop on Virtual Machine Security, pp. 11–22 (2009)

22. Lengyel, T.K., Maresca, S., Payne, B.D., Webster, G.D., Vogl, S., Kiayias, A.: Scalability, fidelity and stealth in the DRAKVUF dynamic malware analysis system. In: Proceedings of the 30th Annual Computer Security Applications Conference, pp. 386–395 (2014)
23. Lindorfer, M., Kolbitsch, C., Comparetti, P.M.: Detecting environment-sensitive malware. In: Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection, pp. 338–357 (2011)
24. Nguyen, A.M., Schear, N., Jung, H., Godiyal, A., King, S.T., Nguyen, H.D.: MAVMM: Lightweight and purpose built VMM for malware analysis. In: Proceedings of the 2009 Annual Computer Security Applications Conference, pp. 441–450 (2009)
25. Otsuki, Y., Takimoto, E., Kashiyama, T., Saito, S., Cooper, E.W., Mouri, K.: Tracing malicious injected threads using Alkanet malware analyzer. IAENG Transactions on Engineering Technologies **247**, 283–299 (2013)
26. Oyama, Y., Kawasaki, Y., Takahashi, K.: Checkpointing an operating system using a parapass-through hypervisor. Journal of Information Processing **23**(2), 132–141 (2015)
27. Pektaş, A., Acarman, T.: A dynamic malware analyzer against virtual machine aware malicious software. Security and Communication Networks **7**(12), 2245–2257 (2014)
28. Spensky, C., Hu, H., Leach, K.: LO-PHI: low-observable physical host instrumentation for malware analysis. In: Proceedings of the 23rd Annual Network and Distributed System Security Symposium (2016)
29. Wang, G., Liu, C., Lin, J.: Transparency and semantics coexist: When malware analysis meets the hardware assisted virtualization. In: Proceedings of the International Standard Conference on Trustworthy Distributed Computing and Services, pp. 29–37 (2013)
30. Zhang, F., Leach, K., Stavrou, A., Wang, H., Sun, K.: Using hardware features for increased debugging transparency. In: Proceedings of the 36th IEEE Symposium on Security and Privacy, pp. 55–69 (2015)
31. Singh, A., Bu, Z.: Hot knives through butter: Evading file-based sandboxes. Tech. rep., FireEye (2014)
32. Brengel, M., Backes, M., Rossow, C.: Detecting hardware-assisted virtualization. In: Proceedings of the 13th International Conference on Detection of Intrusions and Malware and Vulnerability Assessment, pp. 207–227 (2016)
33. Cuckoo Sandbox. `https://cuckoosandbox.org/`
34. OPSWAT: Windows Anti-malware Market Share Reports. `https://www.metadefender.com/stats/anti-malware-market-share-report#!/` (2016)
35. Wyke, J.: Duping the machine – malware strategies, post sandbox detection. In: Proceedings of the 24th Virus Bulletin International Conference, pp. 91–97 (2014)