Regular Paper

API Chaser: Taint-Assisted Sandbox for Evasive Malware Analysis

Yuhei Kawakoya[1,a)] Eitaro Shioji[1,b)] Makoto Iwamura[1,c)] Jun Miyoshi[1,d)]

Received: May 29, 2018, Accepted: December 4, 2018

Abstract: We propose a design and implementation for an Application Programming Interface (API) monitoring system called *API Chaser*, which is resistant to evasion-type anti-analysis techniques, e.g., stolen code and code injection. The core technique in API Chaser is *code tainting*, which enables us to identify precisely the execution of monitored instructions by propagating three types of taint tags added to the codes of API, malware, and benign executables, respectively. Additionally, we introduce *taint-based control transfer interception*, which is a technique to capture precisely API calls invoked from evasive malware. We evaluate API Chaser based on several real-world and synthetic malware to demonstrate the accuracy of our API hooking technique. We also perform a large-scale malware experiment by analyzing 8,897 malware samples to show the practical capability of API Chaser. These experimental results show that 701 out of 8,897 malware samples employ hook evasion techniques to hide specific API calls, while 344 malware ones use target evasion techniques to hide the source of API calls.

Keywords: Malware, Taint Analysis, Anti-analysis, Evasion, Windows API

1. Introduction

Malware threats have become a serious problem on the Internet over the past decade. Malicious activities on the Internet such as massive spam-emailing and denial-of-service attacks have arisen from botnets comprising countless malware-infected machines. To combat malware, analysts utilize various techniques and tools to reveal details of malware activities.

Dynamic analysis is a major technique for malware analysis. Application Programming Interface (API) monitoring, which is a dynamic analysis technique, is especially effective and efficient for rapidly understanding malware activities because an API has rich semantic information. Therefore, it is often used in many research and industrial products such as malware detection [17] and automatic signature generation [44] as an essential component to fight against rapidly evolving malware attacks. That is, API monitoring has become an important technique for both research and industrial security communities.

However, since malware developers are now familiar with malware analysis techniques, they embed anti-analysis functions into their malware to evade API monitoring [6], [7], [26], [29], [30], [31], [40], [49], [56]. Various anti-analysis techniques that evade API monitoring have currently been adopted in malware in the wild. These techniques are mainly classified into two types: hook evasion and target evasion. Hook evasion is a technique to evade hooks added to the entry of APIs for monitoring. Target evasion is used to obfuscate the caller instance of APIs by, for example, invoking APIs from code injected into a benign process. These anti-analysis techniques have become a serious issue for anti-malware research, especially for practical malware analysis systems. However, this issue has not been extensively discussed. As a result, existing API monitors yield opportunities for malware to evade their monitoring.

In this paper, we focus on this issue and present a practical API monitor called *API Chaser*, which is resistant to various evasion-type anti-analysis techniques. API Chaser is built on a whole system emulator, QEMU [5] (actually Argos [41]), and executes monitored malware in a guest operating system (OS) running on it. In API Chaser, we use a *code tainting* technique to identify precisely the execution of monitored instructions. The core idea of code tainting is that we first prepare a taint tag targeted for a specific analysis purpose and add the tag to the target instructions before executing them. Then, we begin to run the executable file containing the monitored instructions. At the virtual CPU of an emulator, we confirm whether or not a fetched instruction contains the taint tag targeted for analysis. If it does, it is executed under analysis. If not, it is executed normally, i.e., it is outside the scope of the monitoring.

The code tainting technique itself is already introduced in Ref. [23], so it is not a new technique. A new aspect of this paper is that we apply the code tainting technique to API monitoring. We call this technique *taint-based control transfer interception*. This technique works as follows. We use three types of taint tags for three different types of instructions: the instructions for APIs, those for malware, and those for benign programs. First, we add

1 NTT Secure Platform Laboratories, Musashino, Tokyo 180–8585, Japan
a) kawakoya.yuhei@lab.ntt.co.jp
b) shioji.eitaro@lab.ntt.co.jp
c) iwamura.makoto@lab.ntt.co.jp
d) miyoshi.jun@lab.ntt.co.jp

This is an extended version of a paper published in the Proceedings of Research in Attacks, Intrusions and Defenses, RAID2013, Lecture Notes in Computer Science, Vol.8145, Springer, Berlin, Heidelberg.

the three types of taint tags to the respective target instructions. Then, when the CPU fetches an instruction and it has a taint tag for the API, it confirms which type of taint tag the caller instruction has. There are three cases: a taint tag for malware, one for benign, and one for API. Each case respectively corresponds to the following situations: an API call from malware, that for a benign process, and that for another API (nested call). Our monitoring target is the call only from malware and we exclude the others.

Taint-based control transfer interception is resistant to evasion techniques because it is able to distinguish between the target instructions and others at byte granularity even when they exist in the same process memory space. In addition, this technique is able to track the movement of monitored instructions by propagating taint tags attached to them when malware injects a malicious code into another process. This technique is independent of OS semantic information such as virtual addresses, Process ID (PID) or Thread ID (TID), and file names. Therefore, it is no longer influenced by the changes in these types of information by malware for evading analysis systems.

In API Chaser, there are also several unique implementations for enhancing the resistance against anti-analysis techniques, i.e., *pre-boot disk tainting* and *code taint propagation*, and for improving the practical capability for large-scale analysis, i.e., *hot-boot* and *one-time disk image*. These techniques contribute to achieving precise API monitoring and a practical malware analysis sandbox, respectively. In the proposed API Chaser implementations, we use 32-bit Windows XP and 7 as the guest OS. However, we do not limit API Chaser to only these two platforms. We believe that the API Chaser design is neutral and we can apply it to other platforms such as a 64-bit Windows 8 or 10 as the guest OS while following the same design.

To show the effectiveness of API Chaser, we conducted several experiments using real-world malware with a wide range of anti-analysis techniques. We evaluated the API monitoring accuracy by analyzing some malware on API Chaser and in comparative environments in which APIs are monitored using existing techniques. Then we compared the logs output by each environment. These experimental results indicate that API Chaser is able to capture precisely the API calls from all sample malware without being evaded. We also evaluated API Chaser with several synthetic malware, in which state-of-the-art evasive techniques were implemented. The experimental results show that API Chaser is sufficiently robust against new emerging techniques such as Process Hollowing [29], AtomBombing [30], PowerLoaderEx [6], Shim-based DLL Injection [40], and Stealth Loader [26].

Moreover, we analyzed 8,879 malware samples collected from the Internet for a large-scale experiment. The samples were classified into 421 malware families with AVClass [45]. Through the analyses, we found 701 hook-evasive ones, which belonged to 36 families, while we found 344 target-evasive malware samples, which belonged to 84 families. We consider that these numbers allow us to argue that hook evasion and target evasion techniques are actually major techniques and widely used among real-world malware.

The first version of this paper was published in 2013 [25]. We have mainly two advances with this paper, compared to the first one. First is that we evaluated API Chaser with five latest code injection techniques, which emerged after 2013. So, these techniques represent new techniques for API Chaser. Since we successfully analyzed them with API Chaser without any updates on its design from the one proposed in the first paper, we can highlight that the design of API Chaser is strong enough to withstand not only existing evasion techniques but also new emerging ones in the future. Second is that we added two new functions to API Chaser, i.e., hot-boot and one-time disk image, to improve the capability of analyzing large-scale malware samples. We evaluated API Chaser with 8,897 real-world malware samples and then showed that API Chaser is capable of analyzing them successfully in practical time.

In summary, we make the following contributions in this paper.

- First, we introduce the API monitoring technique *taint-based control transfer interception* using *code tainting*. It enables correct identification of API calls even from malware using evasion-type anti-analysis techniques.
- Second, we present the implementation details of API Chaser. We also describe the considerations behind each design decision and explain several techniques enabling us to achieve high practicability of API Chaser, which includes pre-boot disk tainting, code taint propagation, hot-boot, and one-time disk image.
- Third, we show the evaluation results of API Chaser using real-world malware and synthetic malware. These malware contain various anti-analysis techniques related to evading API monitoring. The results showed that API Chaser is able to capture correctly APIs called from these malware.

2. Evasion Techniques

In this section, we describe several anti-analysis techniques used in malware for evading API monitoring, and we explain problems facing existing analysis techniques against them. We categorize evasion-type anti-analysis techniques into two types depending on their purpose. The first is hook evasion, which is used for evading API hooks. The second is target evasion, which is used to obfuscate the API caller instances.

2.1 Hook Evasion

Hook evasion is a technique to evade being monitored by an analysis system. We explain five hook-evasion techniques: stolen code, sliding call, copied API obfuscation, name confusion, and Stealth Loader.

Figure 1 (a) shows the behavior of stolen code. Stolen code copies some instructions from the entry of an API to allocated memory areas in the malware process at runtime. When malware attempts to call the API, it first executes the copied instructions and then jumps to the address of the instruction in the API following the copied instructions. Some existing API monitors [4], [38], [48], [53] identify their target API calls by the execution of the instructions at the virtual addresses where these APIs are expected to be located. The expected addresses are computed from the base address of the loaded module contain-

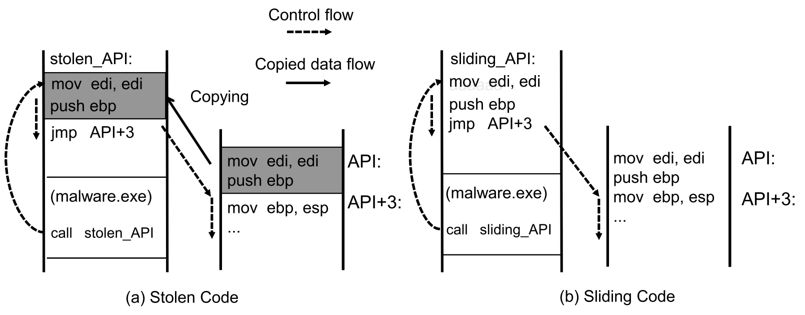


Fig. 1 Stolen Code and Sliding Call Mechanism.

ing these APIs and the offsets to them, which are written in the Portable Executable (PE) header of the module. If the instructions of these APIs are copied to addresses different from the expected ones, existing API monitors may miss capturing the execution of these APIs.

Figure 1 (b) shows the behavior of sliding call. Sliding call behaves almost in the same manner as stolen code. The difference is that malware originally has a few instructions of the entry of a specific API in its body and calls the API after executing those instructions. Almost all existing API monitors focus on the entry of each API [4], [38], [48], [53], [55] as a place to add a hook. This allows the monitoring to be evaded because the instruction at the head of the API is not executed nor even touched by malware using sliding call.

Copied API Obfuscation [49] is an evolved version of stolen code. It copies all instructions of an API to an allocated memory area in the malware process at runtime. Unlike stolen code, copied API obfuscation does not transfer the execution to the instructions of the copied API, i.e., it does not execute them at all. So, if analysis tools add a hook to the entry of an API, they fail to capture the API calls since the entry instruction is not executed at all.

Name confusion involves copying a system dynamic link library (DLL) to another file path while changing its file name. The copied DLL exports the same functions as the original DLL, so the malware loading the copied DLL can still call the same functions as those in the original DLL. If the name has been changed, some analysis systems [4], [38], [48], [58] that depend on the names of the module to identify their target can be evaded. In addition, name confusion is also often used for target evasion, e.g., malware changes its name to that of a system executable file installed as default such as svchost.exe or winlogon.exe.

Stealth Loader was introduced by Kawakoya et al. [26] for evading existing static and dynamic analysis tools. It is a program loader that loads Windows system DLLs such as kernel32.dll or ntdll.dll without leaving any trace to be detected. The program loader is embedded into a protected executable file and begins to run before its original code is executed. By loading a system DLL with Stealth Loader, the loaded DLL is not recognized as 'loaded' by Windows OS and even analysis tools because there is no trace to recognize that the DLL was loaded. Since analysis tools fail to recognize the existence of the loaded DLL, they also fail to capture API calls of the functions exported from the unrecognized system DLL.

2.2 Target Evasion

Target evasion is a technique in which malware attempts to evade being the target of analysis. We explain two target evasion techniques: code injection and file infection.

Code injection injects a piece of malicious code into another process, and enables that code to be executed in that process. If an API monitor distinguishes its monitoring target based on PID or TID, which is very common in most existing systems [4], [38], [48], [53], [58], it needs to add hooks to specific APIs such as WriteProcessMemory or CreateRemoteThread to extract the destination of the injection. The traceability in existing systems is tightly bound to injection techniques. That is, they rely on heuristics to track the movement of their analysis targets. Even if it succeeds in identifying the injected process as a monitoring target, it would be difficult to distinguish correctly APIs called from malicious code injected into the process and those called from the original code in the process.

In this decade, several new code injection techniques such as Process Hollowing [29], AtomBombing [30], PowerLoaderEx [6], and Shim-based DLL Injection [40] have been applied in real-world malware. These techniques avoid using the APIs commonly used for code injection, i.e., CreateRemoteThread, WriteProcessMemory, and VirtualAllocEx. These techniques may evade detection if these systems rely on these API calls to detect the code injection behavior for adding the injected processes to a target process list.

File infection is another target evasion technique. It basically adds a piece of code to an executable file and modifies pointers in its PE header to cause the added code to execute after the program begins to run. Similar to code injection, it is difficult to distinguish between API calls from a malicious code and those from the original benign code if the API monitor tries to identify its target using PIDs or TIDs.

3. Proposed Approach

To address the evasion problems that existing API monitors have, we propose the *taint-based control transfer interception* API monitoring technique that uses *code tainting* to identify precisely the execution of APIs. First, we define some terms and the scope of this paper. Second, we present code tainting. Third, we describe the types of monitored instructions. Fourth, we present taint-based control transfer interception, i.e., how to capture API calls invoked from malware and exclude the ones invoked from benign processes and nested API calls.

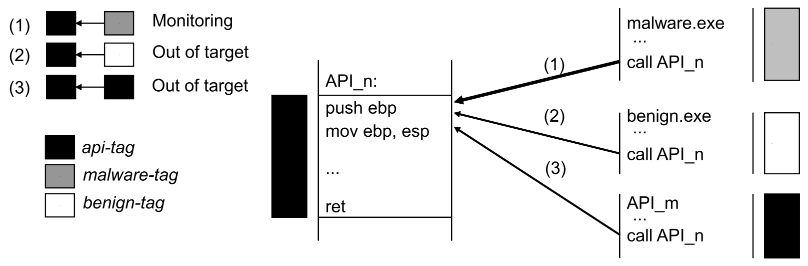


Fig. 2 Taint-based control transfer interception.

3.1 Definitions and Scope

We define three important terms used in this paper: API, API call, and API monitoring.

- *API* is a function comprising more than one instruction to conduct a specific purpose and we use it interchangeably with a user-land Windows API, which is a function provided from the Windows operating system and libraries.
- *API call* is a control transfer with valid arguments from an instruction outside of an API to an instruction within the API.
- *API monitoring* is a technique to detect the first execution of an instruction of monitored APIs immediately after control has been passed from an instruction outside of the API.

We explain the scope of this paper. The anti-analysis techniques in the scope are those that were mentioned in the previous section, i.e., those used for hiding API calls that malware has actually invoked. We exclude the anti-analysis techniques designed to use conditional execution to evade analysis systems, e.g., trigger-based ones [8] and stalling code [27], from the scope of this paper. We also exclude the case that malware invokes functions of modules, e.g., DLLs, statically linked to the malware, which do not execute any instructions of system modules that we prepared in our analysis environment.

3.2 Code Tainting

Code tainting is a taint analysis application and a technique used to identify the execution of monitored instructions based on taint tags attached to them. It adds taint tags to the target instructions before executing them. Then, when the CPU fetches an instruction, it confirms if the instruction (actually the opcode of the instruction) has a taint tag. If the instruction has a taint tag targeted for analysis, it will be executed under analysis. If not, it will be executed normally. When monitored instructions are operated as data, taint tags added to the instructions are propagated in the same way as data tainting. That is, we can track the movement of monitored instructions based on the taint tags.

There are three advantages of code tainting for monitoring malware activities. First, it becomes possible to conduct fine-grained monitoring. This property is effective against malware using target evasion techniques. Code tainting can distinguish the target instructions and others at byte granularity based on taint tags, even though there are both injected malicious instructions and benign ones mixed together in the same process space or the same executable file. Second, it allows us to track the movement of the target instruction by propagating taint tags attached to them. This property is effective against both target evasion and hook evasion techniques. For example, when malware injects its malicious code into other processes or other executable files, code tainting can track the injection by propagating taint tags added to the malicious code. Third, it is no longer influenced by changing the semantic information of an OS, e.g., virtual addresses, PID or TID, and file names. This property is also effective against both target evasion and hook evasion techniques such as name confusion or Stealth Loader because it does not depend on these types of semantic information at all for monitoring API calls, and depends solely on taint tags.

A similar technique to code tainting has been used in previous research [35], [41] to detect attacks by tainting received data from the Internet and then monitoring a control transfer to the tainted data. We leverage the technique for malware analysis on API monitoring. The difference is that code tainting adds taint tags to the code with the obvious intention of monitoring its execution, whereas the previous research taints all received data to detect a control transfer to it.

3.3 Tag Types and Monitored Instructions

We use the following three types of taint tags to identify the execution of three types of instructions for API monitoring.

- *Api-tags* target instructions in each API
- *Malware-tag* targets instructions in malware
- *Benign-tag* targets instructions in benign programs

We taint all instructions in each API with api-tags. We use this type of tag to detect the execution of APIs at the CPU. Moreover, we embed API-identifier information in each api-tag which we can use to distinguish the execution of each type of API. Regarding malware-tag, we taint all bytes in a malware executable file and dynamically generated code with malware-tags. We use malware-tags to identify the caller instruction of APIs and detect the execution of malware instructions. On the other hand, we taint all bytes in benign programs with benign-tags. By benign programs, we mean all files that have been installed on Windows by default, or in other words, all instructions except for those in malware and APIs. We mainly use this type of taint tag to identify the caller instruction of APIs and then exclude the API calls from the monitoring target.

3.4 Taint-Based Control Transfer Interception

We use code tainting with the three types of taint tags to monitor APIs invoked from malware. When a CPU fetches an instruction and the instruction has an api-tag, it confirms the taint tag attached to the caller instruction. There are three cases as shown in **Fig. 2**: the API is called from malware, a benign process, and the

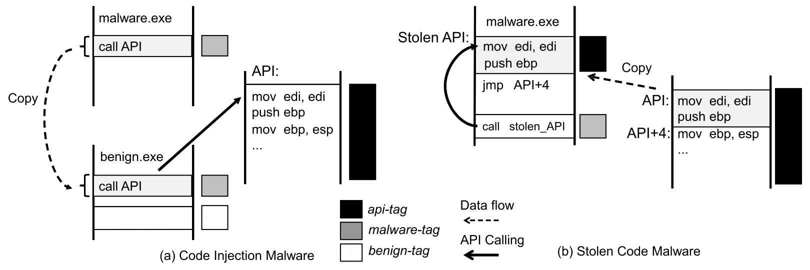


Fig. 3 Taint-based control transfer interception against anti-analysis.

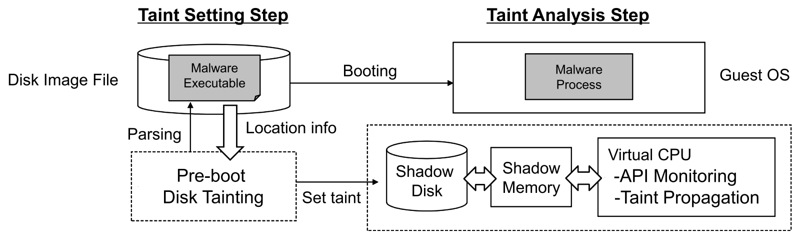


Fig. 4 Analysis process for API Chaser.

internal of other APIs (nested call). As for the first case, shown in Fig. 2 (1), if the caller instruction has a malware-tag, it determines that the API call is from malware. Thus, it captures the API call and collects the information related to the API call such as its arguments. With regard to the second, shown in Fig. 2 (2), if caller one has a benign-tag, it determines that the API call is from a benign process. Thus, it is outside the target monitoring and does not need to capture this API call. As for the third, shown in Fig. 2 (3), if the caller has an api-tag, it is a nested API call. Nested API calls are also excluded from the monitoring target, so that we can focus only on API calls directly invoked from malware. This makes the behaviors of malware clearer and easier to understand.

In **Fig. 3**, as a running example, we explain the behaviors of taint-based control transfer interception against the two anti-analysis techniques: code injection and stolen code. Figure 3 (a) shows the behavior against code injection. When malware injects code from malware.exe to benign.exe, the taint tags of the code are propagated. The API call from the injected code is a control transfer from an instruction with a malware-tag to an instruction with an api-tag. Then, we can identify it as our target API call. On the other hand, Fig. 3 (b) shows the behavior of calling a stolen API. When few instructions at the entry of the API are copied to the allocated memory area in malware.exe, the taint tags added to the instructions are also propagated. The call instruction, `call stolen_API`, has a malware-tag and the copied instruction, `mov edi, edi`, has an api-tag, so we detect the API call and include it in the monitoring target.

4. System Description

In this section, we present an overview of *API Chaser*, which uses taint-based control transfer interception for monitoring API calls. First, we briefly explain the main components of API Chaser. Second, we illustrate its malware analysis process. Third, we present the enabling techniques used in API Chaser: *pre-boot disk tainting* and *code taint propagation*.

4.1 Components

API Chaser is built on a whole system emulator, QEMU (actually on Argos). API Chaser has the following components: virtual CPU for API monitoring and taint propagation, shadow memory to store taint tags for virtual physical memory (hereafter "physical memory"), and shadow disk to store taint tags for a virtual disk (hereafter "disk").

A virtual CPU is the core component of API Chaser. It is a dynamic binary translator that translates a guest instruction to host native instructions. With dynamic binary translation, it conducts API monitoring as mentioned in the previous section and taint propagation based on our propagation policy, which is explained in a later subsection. The shadow memory is a data structure for storing taint tags added to data on physical memory. When the virtual CPU fetches an instruction, it retrieves the taint tag added to the instruction from the shadow memory. The shadow disk is also a data structure for storing taint tags added to data on a disk. When data with taint tags are written to a disk, the taint tags are transferred from the shadow memory to the shadow disk and stored in the corresponding entries of the shadow disk. When transferring data with taint tags from a disk to physical memory, the taint tags are also transferred from the shadow disk to the shadow memory.

4.2 Analysis Process

Figure 4 shows the analysis process for API Chaser. There are two steps for API Chaser to analyze malware: taint setting and analysis.

4.2.1 Taint Setting Step

In the taint setting step, API Chaser conducts pre-boot disk tainting, which adds taint tags to all the target instructions in a disk image file before booting a guest OS. The details of pre-boot disk tainting are given in the following subsection.

4.2.2 Analysis Step

In the analysis step, API Chaser first boots the guest OS installed on the disk image file. During the boot, target files containing target instructions are loaded onto physical memory. At

	Rule1 and Rule2	Rule3
Target Instruction	mov [edi], eax	call CryptEncrypt(,,, pbData, pdwDataLen,,,);

Code Taint Propagation Handling Code	if eax is tainted:
	    set the tag of eax on [edi];
	else:
	    if 'mov' has a malware-tag
	        set a malware-tag on [edi];
		if 'call' has a malware-tag:
		    for(i = 0; i < *pdwDataLen; i++) {
		        set a malware-tag on pbData[i];
		    }

Fig. 5 Code taint propagation example.

the same time, the taint tags added to the target instructions are also transferred from the shadow disk to the shadow memory. After completing the boot, API Chaser executes malware and initiates analysis. During the analysis, API Chaser conducts API monitoring and taint propagation based on our policy.

4.3 Enabling Techniques

We explain the enabling techniques used in API Chaser to support the API monitoring: pre-boot disk tainting and code taint propagation.

4.3.1 Pre-boot Disk Tainting

Pre-boot disk tainting is a technique that adds taint tags to target instructions on a disk image file before booting a guest OS. Properly adding taint tags to all target instructions is not an easy task because they may be copied and widespread over the system after a guest OS has booted. For example, after booting a guest OS, an API instruction may be on a disk, loaded onto memory, swapped out to disk, or swapped into memory. When we add taint tags to a target instruction, we must identify all the locations of widespread instructions and add tags to all of them. If we miss adding a tag to any one of them, it may allow malware to evade the API monitoring.

To avoid this troublesome task, we use pre-boot disk tainting. The procedure is given hereafter. First, it parses a disk image file containing target instructions and identifies the location where the target instructions are stored. We use disk forensic tools [10] to identify files containing target instructions, and then, if necessary, we acquire the offsets of the target instructions from the PE header of the files and identify the locations of each API using disassemble tools [20], [21]. Second, it adds taint tags to the corresponding entries of a shadow disk based on the calculated location. Before launching a guest OS, all instructions surely reside on a disk and they are not widespread yet. Pre-boot disk tainting simplifies the tainting task because only target instructions on a disk require attention. We no longer need to care whether or not target instructions have been loaded.

4.3.2 Our Taint Propagation Policy

API Chaser conducts taint propagation based on pre-defined rules. The pre-defined rules are mainly composed of two types: basic rules and rules for code taint propagation.

Basic rules are defined based on each instruction type, such as data-move, unary arithmetic, or binary arithmetic operations. We basically use the rules of Argos [41] for API Chaser as they are. More concretely, when API Chaser handles a data-move operation, such as `mov`, it propagates its taint tag to the destination if the source operand is tainted. When it handles an unary arithmetic operation, such as `inc`, it preserves the tag of the operand as if the operand has a taint tag. When it handles a binary arithmetic operation, such as `add`, it propagates the tag of the source operand to the destination if any one of the source operands is tainted. If both operands are tainted, it propagates the tag of the first operand. However, this propagation rule may cause it to disconnect a taint propagation with the tag of the second operand because we have to discard the tag of the second operand even when the tag plays an important role for tracking a specific data-flow. Regarding this problem, we discuss it in Section 8.2.

In addition to the above rules, we use our original taint propagation rules for memory-write operations called *code taint propagation* to prevent malware from avoiding code tainting by generating a code using implicit-flow-like code extraction. Implicit flow is a process where a value with a taint tag affects the decision making of the following code flow. However, there is no direct dependency between the value and other values operated in the following code. Thus, a taint tag is not propagated over the implicit flow, even though they are semantically dependent on each other. It is reported that taint tags are not properly propagated in some Windows APIs that use implicit-flow-like processing [58]. Actually, we observed some cases in which malware-tags added to the code of malware were not propagated to its dynamically generated code. This is because most obfuscated malware has encrypted or compressed original code in its data section and it uses implicit-flow-like data-processing to unfold compressed or encrypted code and extract its original code. If we fail to propagate malware-tags properly, we miss identifying the execution of malware instructions.

To address this, we use code taint propagation for code dynamically generated by malware. Code taint propagation has the following rules.

- **Rule1**: If an executed instruction is tainted with a malware-tag and its source operand is not tainted, the taint tag of the instruction, i.e., malware-tag, is added to the destination operand.
- **Rule2**: If an executed instruction is not tainted or tainted with the other tags, it does not propagate the taint tag of the instruction to its destination.
- **Rule3**: If an instruction calling an API is tainted with a malware-tag, the taint tag of the instruction, i.e., malware-tag, is added to the written data by the API.

The bottom-left pseudocode in **Fig. 5** is an example of **Rule1** and **Rule2**, illustrating the case of `mov [edi], eax`. If the source operand of the target instruction, eax, does not have any taint tags and the opcode, `mov`, has a malware-tag, we add malware-tags to the destination operand, `[edi]`. Consequently, it appears as if it propagates taint tags of opcode to the destination operand of the opcode. The bottom-right pseudocode in Fig. 5 is an example of **Rule3**, illustrating the case of `call`

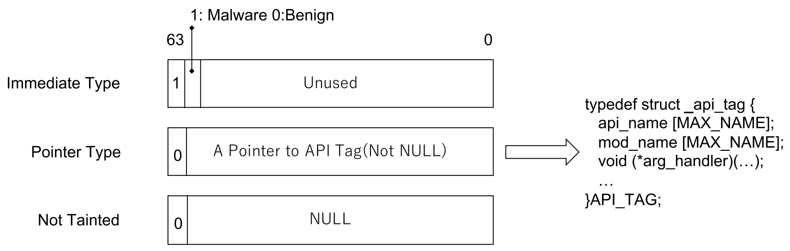


Fig. 6 Taint tag format.

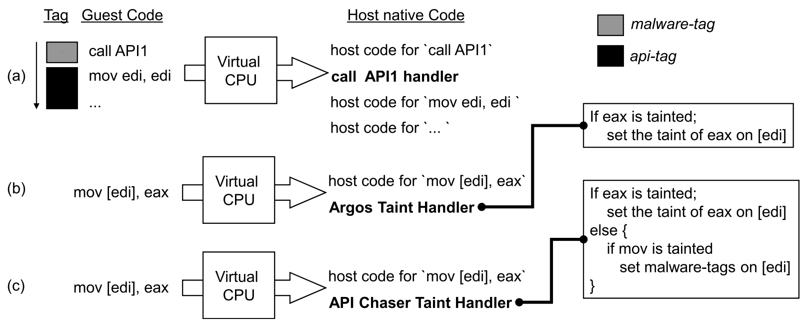

Tag
Guest Code
Host native Code
malware-tag
api-tag
(a)
call API1
mov edi, edi
...
Virtual CPU
host code for `call API1`
call API1 handler
host code for `mov edi, edi `
host code for `... `
If eax is tainted; set the taint of eax on [edi]
(b)
mov [edi], eax
Virtual CPU
host code for `mov [edi], eax`
Argos Taint Handler
If eax is tainted; set the taint of eax on [edi] else { if mov is tainted set malware-tags on [edi] }
(c)
mov [edi], eax
Virtual CPU
host code for `mov [edi], eax`
API Chaser Taint Handler

Fig. 7 Examples of dynamic binary translation.

CryptEncrypt whose prototype is shown below. The call instruction has a malware-tag and it calls CryptEncrypt API, which is a function that encrypts the passed data and writes its output to the memory area pointed to by the argument, pbData. The argument, pdwDataLen, indicates the size of the output data.

```
BOOL WINAPI
  CryptEncrypt(
          _In_ HCRYPTKEY hKey,
          _In_ HCRYPTHASH hHash,
          _In_ BOOL Final,
          _In_ DWORD dwFlags,
          _Inout_ BYTE *pbData,
          _Inout_ DWORD *pdwDataLen,
          _In_ DWORD dwBufLen);
```

We detect the moment when execution is returned from the API by monitoring a control transfer from an instruction with the api-tag to one with the malware-tag, and then add malware-tags to the written bytes by acquiring the location of the written bytes from pbData. It seems as if the taint tag of the call instruction is propagated to the written bytes of the API called from the instruction. Owing to code taint propagation, we can taint all generated codes with malware-tags and identify the execution of the code based on its taint tags. We will discuss the side-effects of code taint propagation in Section 8.3.4.

5. Implementation

In this section, we explain the details of the API Chaser implementation focusing on extensions from Argos [41] and techniques for making API Chaser practical for industrial use-cases. We present the taint tag format, virtual CPU, shadow memory, shadow disk, virtual direct memory access (DMA) controller, API argument handlers, hot-boot, and one-time disk image.

5.1 Taint Tag Format

We introduce the format of a taint tag stored in the shadow memory and shadow disk. The size of a taint tag is eight bytes. There are three format types, as shown in **Fig. 6**: immediate format type for malware-tags and benign-tags, pointer format type for api-tags, and not-tainted type. The format is chosen depending on the type of taint tag. We distinguish the format type based on the highest bit of a tag. In the case of the immediate type, we distinguish malware tags from benign tags based on the second highest bit. API Chaser uses only the highest two bits, and the other bits are unused. On the other hand, in the case of the pointer type, a taint tag is a pointer to an API tag data structure. An API tag structure is a data structure that stores information related to an API such as the API name, DLL name, and API argument handling functions. We create an API tag data structure for each API, and all instructions in each API have a taint tag with a pointer to the same API tag data structure.

5.2 Virtual CPU

The virtual CPU of QEMU (Argos) achieves virtualization with dynamic binary translation. It translates instructions from a guest OS to instructions for a host OS to emulate consistently the guest OS on the host OS. Argos adds a taint tracking mechanism to the dynamic binary translation. That is, it propagates taint tags from source operands to the destination based on its taint propagation policy after executing each instruction. In API Chaser, we added two new functions to the virtual CPU: an API monitoring mechanism and code taint propagation.

Figure 7 (a) shows the API monitoring mechanism of API Chaser in the virtual CPU. When an API call is invoked from malware, i.e., the execution transferring the instruction with a malware-tag to that with an api-tag, the virtual CPU retrieves the information related to the API through its API tag data structure pointed to by the taint tag, and generates host native instructions for handling the API, i.e., invoking the API handler function. An API handler outputs the API name and DLL name, and internally invokes argument handling functions.

As for code taint propagation, Figs. 7 (b) and 7 (c) show the difference in the behaviors between Argos and API Chaser. In the case of Argos, when it reads a guest OS instruction for writing memory, it generates a taint handling function as host native code. The function propagates taint tags from source operands to the destination if the source has any taint tags. In the case of API Chaser, it generates its original taint handling function for code taint propagation. The function adds malware-tags to the writing destination if the source operand does not have any taint tags and the opcode has a malware-tag.

5.3 Shadow Memory, Disk, and Virtual DMA Controller

Shadow memory is an array of eight-byte entries where each entry corresponds to a byte on physical memory. Argos originally has shadow memory, but it has only a one-byte taint tag space for one byte on the physical memory, which is only used for determining the taint state. We extend it to an eight-byte taint tag space for one byte to store a pointer to the API tag data structure. Therefore, we need memory space eight times as large as the physical memory for the shadow memory. For example, if the size of the physical memory is 256 Mbytes, the size of the shadow memory is 2 Gbytes. Considering large-scale analysis, we design the shadow memory to be dynamically allocatable at the time when it becomes necessary in order to suppress the increase in simultaneous memory consumption when running multiple API Chaser instances at the same time.

The shadow disk is a binary-tree data structure for storing taint tags added to data on a disk. The entries for the structure contain information related to tainted data on a disk such as the sector number, offset, size, taint tag buffer, and pointers represented by the nodes of the binary tree. A taint tag entry for one-byte data on a disk takes eight bytes of space, so we need eight times as large a memory space as a disk for the shadow disk. However, the size of the disk is much larger than that of the physical memory, so it is difficult to allocate sufficient memory space to store the taint tags of all data on a disk beforehand. Thus, we design the memory space for the shadow disk to be dynamically allocated as needed. Argos does not have a shadow disk, so we newly implemented it for API Chaser.

In API Chaser, the virtual DMA controller transfers taint tags between the shadow memory and a shadow disk. API Chaser monitors DMA commands at the virtual DMA controller, and when it finds a request for transferring data, it acquires the data location from the request and confirms whether or not the transferred data has taint tags. If it does, the virtual DMA controller transfers the taint tags between the shadow memory and shadow disk. Argos does not have this mechanism either, so we newly implemented it for API Chaser as well.

5.4 API Argument Handler

To obtain more detailed information of API calls, we extract argument information passed to them when they are called and when the execution is returned from them. To do this, we prepare an API argument handler for each API. We extract the argument information such as the number of arguments, variable types, size, and whether it is an input or output argument from the Windows header files provided by the Windows software development kit (SDK). For undocumented APIs, we extract their information from the web site [37] and source code of React OS [43]. We register an API argument handler to an API tag data structure when we create the data structure for adding api-tags to the instructions of each API. The handler is invoked from the virtual CPU when it detects an API call invoked from malware and outputs detailed argument information related to the API into a log file.

5.5 Hot-boot

We use the snapshot capability of QEMU for hot-boot, which skips the boot process of a guest OS and enables quick initialization of analysis. Taint analysis provides deep insight into malware behavior by adding data-flow analysis. However, one drawback to taint analysis is performance degradation as measured in Section 6.5. In our brief experiment, it took more than 10 mins to boot a guest OS on API Chaser from the cold disk image, i.e., cold-boot, and be ready for analysis. This performance penalty may become a bottleneck for API Chaser for use in industrial use-cases considering that an anti-virus company reportedly collects a plethora of large-scale malware samples per day from the Internet or their customers.

To compensate for such a performance penalty, we implemented a hot-boot capability using `savevm` and `loadvm` QEMU Monitor Protocol (QMP) commands, which save the state of a guest OS running on QEMU and restore that state, respectively. We extended these two commands with a capability for handling shadow memory. That is, the hot-boot capability stores the state of the shadow memory when `savevm` is executed by taking a snapshot and restores the state when `loadvm` is issued for a guest OS to restart running from the taken snapshot. With the extended `savevm` and `loadvm` commands, we achieve hot-boot in API Chaser. This capability allows us to initiate an analysis immediately after launching API Chaser and makes it practical when analyzing large-scale malware.

We can use hot-boot and pre-boot tainting at the same time without compromising any advantages from pre-boot disk tainting. When the `savevm` command is issued after a guest OS has been booted and ready for analysis, the API code, which is tainted with api-tags, may exist in both the memory and disk. To handle this situation, when we take a snapshot, we can save the status of the shadow memory using the extended `savevm`. When we load the taken snapshot, we can restore the status of the shadow memory with the extended `loadvm` command for the API code on memory and perform pre-boot disk tainting to taint the API code on the disk.

5.6 Parallel Analyses

To analyze large-scale malware samples, it is natural to run multiple instances in parallel. For that purpose, we must reduce the simultaneous consumption of memory and disks because the physical resources such as memory or disks are limited on a machine. To reduce the memory consumption, we developed dynamic shadow memory and disk allocations as explained in Section 5.3. Additionally, to reduce disk space consumption, we developed a one-time disk image. We explain this in this subsection.

5.6.1 One-time Disk Image

Disk space is a physically-limited resource. So, one requirement for API Chaser as a practical analysis environment is to reduce the consumption of disk space to run multiple API Chaser instances concurrently. Another requirement for API Chaser is that it must have a capability for returning to a clean state after one analysis has completed because the environment may be destroyed or compromised by the malware under analysis. One option for this requirement with QEMU is to use a `-snapshot` option, which keeps the original disk read-only and redirects all disk-writes to a temporal disk. However, unfortunately, QEMU version 1.1.50 on which Argos was built does not support this `-snapshot` option for a guest OS booted with `loadvm` [15].

To satisfy these two requirements at the same time, we developed a one-time disk image for API Chaser. The one-time disk image is an extended qcow2 image file format [34]. We leverage the backing_file capability of the qcow2 image format to retain the clean state. The one-time disk image works as follows. First, we cold-boot a guest OS with the `-snapshot` option and take a snapshot with `savevm` after the booting has completed and is ready for analysis. Due to the `snapshot` option, QEMU creates a temporal disk image in the `tmp` directory. Second, we configure the backing_file option of the temporal disk image with the original disk image. This configuration allows read-access to the data that do not exist in the temporal disk image to be redirected to the original disk image. In addition, this configuration allows write-access to be routed to the temporal disk image.

Using this capability, we retain the read-only nature of the original disk image and redirect all write access to the temporal disk image even after a guest OS is restarted with the `loadvm` command. When we begin analysis, we simply copy the configured temporal disk image, rename it, and boot a guest OS from the copied temporal disk image as an analysis environment of API Chaser. After analysis is complete, we simply discard the copied temporal disk image and restart a new analysis by copying a new temporal disk image from its original. This approach allows us to prepare only one original disk image and multiple copies of the temporal disk image for multiple analysis environments. Since the size of a temporal disk image is much less than that of the original disk image, we can reduce the consumption of disk space for preparing multiple disks for multiple instances. Additionally, we can retain the clean environment state for each analysis.

6. Experiments

To show the effectiveness of API Chaser, we conducted four experiments to evaluate the accuracy, the analysis capability for new emerging anti-analysis techniques, the capability for large-scale analysis, and the performance of API Chaser.

6.1 Experimental Environment

All experiments were conducted on a computer with Intel Xeon CPU E5-1650 v4 3.6 GHz, 64 G memory and 360 G SSD. API Chaser runs on Ubuntu Linux 14.14, and the guest OS was Windows XP Service Pack 3 (WinXPsp3) or Windows 7 Service Pack 1 (Win7sp1). The guest OS was allocated 256 Mbytes for its physical memory in the case of WinXPsp3 and 1 Gbyte in that of Win7sp1. We targeted 6,862 APIs in major Windows system DLLs for monitoring.

6.2 Accuracy Experiments

We evaluated API Chaser from the viewpoint of its resistance against hook evasion and target evasion. We prepared several malware executable files that have various anti-analysis functions and we used them to evaluate the resistance of API Chaser against hook evasion and target evasion. As a comparative environment, we prepared two different implementations of API Chaser that respectively use existing techniques to detect API calls (Type I) or identify target code (Type II). In each experiment, we executed some malware on API Chaser and one of these comparative environments for five minutes, acquired API logs that were respectively output by the two environments, and then compared them. When there were some differences between the logs, we revealed the causes of the differences by manually analyzing malware and investigating the infected environment using IDA [20] and The Volatility Framework [50] to determine whether the fault was in API Chaser or in the comparative environments. We used WinXPsp3 as a guest OS of API Chaser for the experiments.

6.2.1 Hook Evasion Resistance Experiment

We used four real-world malware samples which have hook evasion functions with stolen code, sliding call, or API hooking. We include malware using API hooking into the samples for this experiment because the behavior of API hooking is similar with the one of stolen code and it is also able to evade API monitoring as stolen code does. Regarding the other two hook evasion techniques introduced in Section 2, i.e., name confusion and copied API obfuscation, we could not find any malware sample using them in the wild. So, we qualitatively discuss the resistance capability of API Chaser against these two techniques in Section 8.1.

With the four malware samples, we executed them on both API Chaser and a comparative environment (Type I). Type I is another implementation of API Chaser with a different technique to detect API calls. It detects API calls by comparing an address pointed to by an instruction pointer to addresses where APIs should reside, which is a common existing technique. The other components of Type I are the same as API Chaser.

6.2.1.1 Results

Table 1 lists the results of this experiment. We manually investigated the causes of the differences in captured API calls and revealed that all of them were caused by false negatives of Type I. We explain the details of the two cases, Themida and Mystic!gen2, although the others also had the same reason for their differences. In the case of Themida, API Chaser captured 2,966 more API calls than Type I. All the unmatched API calls were detected in the dynamically allocated and writable memory area. On the other hand, all the matched API calls were detected in the memory area where system DLLs were mapped. We manually confirmed that all API calls, except for API calls with no arguments that API Chaser detected, had valid argument information. Thus, these were not false positives of API Chaser, but false negatives of Type I. As we mentioned, API Chaser can detect the stolen API call by propagating taint tags added to an API

Table 1 Results of Hook Evasion Resistance Experiment.

Virus Name	API Chaser	Type I	Unmatched	Reason	Anti-analysis
Win32.Virut.B	6,361	4,852	1,509	F.N. of Type I	API Hook
Themida	43,994	41,028	2,966	F.N. of Type I	Stolen Code
Infostealer.Gampass	38,382	1,397	37,485	F.N. of Type I	Sliding Call
Packed.Mystic!gen2	97,364	97,363	1	F.N. of Type I	Sliding Call

Themida: calc.exe packed by Themida [51]. F.N.: False Negative.

Table 2 Results of Target Evasion Resistance Experiment (Tracking).

Virus Name	Description of Anti-analysis Behavior	Result
Win32.Virut.B	Infecting files with CreateFileMapping	✓
	Injecting code with WriteProcessMemory	✓
Trojan.FakeAV	Injecting code with WriteProcessMemory	✓
	Changing the name of rundll32.exe to jahjah06.exe	✓
Infostealer.Gampass	Injecting code with WriteProcessMemory and the injected code loads a dropped DLL	✓
	Changing its name to svchost.exe	✓
Spyware.perfect	Injecting a dropped DLL with SetWindowsHookEx	✓
Trojan.Gen	Injecting a dropped DLL via AppInit_DLLs registry key	✓
Backdoor.Sdbot	Executing a dropped EXE as a service	✓

✓indicates that API Chaser can correctly track and identify anti-analysis behaviors without being evaded.

to the stolen instructions, while Type I cannot because it does not track the movement of the stolen instructions. This capability contributes to the resistance of API Chaser against hook evasion techniques. In the case of Packed.Mystic!gen2, we confirmed that it used the sliding call technique. The following code snippet is from a sliding call in this malware.

```
0x00408175 push ebp
0x00408176 mov ebp, esp
0x00408178 sub esp, 20h
0x0040817B cmp dword ptr [eax], 8B55FF8Bh
0x00408181 jnz loc_40818C
0x00408187 add eax, 2
0x0040818C add eax, 6
0x00408191 jmp eax ;to API+2 or API+6
```

The `cmp` instruction at 0x0040817B confirms the existence of the following four bytes, 0x8B, 0xFF, 0x55, and 0x8B at the address stored in `eax`, which points to the head of an API. These four bytes may indicate the assembler instructions, `mov edi, edi; push ebp; mov ebp, esp;`, which is a prologue for a hotpatch-enabled API, which has sufficient space for hooking before the first instruction of the API. In fact, the total size of the three assembler instructions is a total of six bytes. If the malware finds these four bytes at the entry of the API, it jumps to a location at six bytes after the entry of the API to avoid monitoring. API Chaser adds taint tags to all instructions in each API, so it was able to detect the execution of the instruction at API entry + 0x6 and identified it as an API call from malware.

6.2.2 Target Evasion Resistance Experiment

We prepared six real-world malware with target evasion functions. Using these malware, we evaluated the following two capabilities of API Chaser: tracking the movement of target code and identifying the target code in a code-injected process or executable file. As for the tracking capability, we confirmed that API Chaser can capture API calls from a process or executable code-injected file by the six malware. In regard to the identifying capability, we prepared another comparative environment (Type II). The Type II environment is different from API Chaser in identifying target code and tracking code injection. It identifies its target depending on the PID and tracks code-injection based on invocation of specific API calls and DLL loading events. For example, Type II hooks the invocations of WriteProcessMemory API calls and extracts the PID of the destination process of the writing from its arguments. Then, it includes the PID into its monitoring targets. The Type II components except for those for identifying and tracking target code are the same as API Chaser.

6.2.2.1 Results

Table 2 lists the results of the tracking experiment. API Chaser successfully tracked all the behaviors of the injected code without being evaded. We consider that Type II can also track them if it knows how target malware evades monitoring and it prepares mechanisms for tracking the behaviors beforehand. However, it is practically difficult to know all code injection methods and prepare for them before executing target malware because there are many unpublished functions in Windows and third party software. On the other hand, API Chaser can track code injection by propagating taint tags added to target malware. Since API Chaser does not depend on individual code injection mechanisms, we can say that it is more generic than the existing approach depending on each injection method for tracking them.

Table 3 lists the results of the identifying experiment. We manually investigated the causes of the unmatched API calls and revealed that all the unmatched API calls were caused from false positives of Type II. That is, API Chaser successfully identified all API calls invoked from an injected code in a benign process and eliminated API calls invoked from the benign part of code in the process. We explain the details of the two specific cases, Trojan.FakeAV and Infostealer.Gampass, although the others also yield the same results. In the case of Trojan.FakeAV, all the matched API calls were invoked from the dynamically allocated memory area which was allocated and written by Trojan.FakeAV, while unmatched API calls were invoked from the memory area where explorer.exe was mapped. This indicates that API Chaser captured the API calls invoked from the code injected by Trojan.FakeAV and Type II additionally captured API calls invoked

Table 3 Results of Target Evasion Resistance Experiment (Code identification).

Virus Name	Injected Process	API Chaser	Type II	Unmatched	Reason
Win32.Virut.B	notepad.exe	315	3,020	2,705	F.P. of Type II
Win32.Virut.B	winlogon.exe	184	783	599	F.P. of Type II
Trojan.FakeAV	explorer.exe	20	1,782	1,762	F.P. of Type II
Infostealer.Gampass	explorer.exe	147,646	149,408	1,762	F.P. of Type II
Spyware.perfect	notepad.exe	4,792	7,511	2,719	F.P. of Type II
Trojan.Gen	notepad.exe	230	3,222	2,992	F.P. of Type II

F.P.: False Positive. We filtered nested API calls by white-listing the memory address ranges where known system DLLs were mapped.

from original code in the code-injected benign process. In the case of Trojan.Gen, all the matched API calls were invoked from tzdfjhm.dll, while all the unmatched calls were from the memory area where notepad.exe was mapped. Library tzdfjhm.dll was registered to the registry key, AppInit_DLLs, which is used by malware for injecting a registered DLL into a process. The DLL was dropped and registered to the key by Trojan.Gen.

6.3 Synthetic Malware Experiment

The purpose of this experiment is to show the feasibility of API Chaser against state-of-the-art evasion techniques including those introduced in academic studies. For that purpose, we collected proof-of-concept (PoC) codes of the following techniques, Process Hollowing [29], [31], AtomBombing [7], [30], PowerLoaderEx [6], Shim-based DLL Injection [40], and Stealth Loader [26]. Then, we generated synthetic malware samples based on the PoC codes and analyzed them with API Chaser. We used Win7sp1 as a guest OS of API Chaser for this experiment. The reason why we focus on these five techniques is that they appeared or became major after our paper was first published in 2013 [25]. So, these techniques represent new techniques for API Chaser and if we can precisely analyze the malware with these techniques with API Chaser, we can demonstrate that the design of API Chaser is possibly strong enough for analyzing future-emerging techniques.

6.3.1 Process Hollowing

Process Hollowing is a technique that hides the presence of a malware process. First, it creates a benign executable file process that is suspended. Next, it overwrites the contents of the suspended process with those of a malicious executable file, and then resumes execution of the process after overwriting is completed. The PoC code for this technique, Ref. [31] creates a process of svchost.exe that is suspended and overwrites its contents with that of the specified (malicious) executable file.

6.3.1.1 Result

API Chaser successfully captured the APIs invoked from the specified executable file overwritten in the process of svchost.exe. API Chaser kept tracking the movement of the overwritten code of the specified executable code by propagating tags added to the code so that APIs invoked from the code were the control transfers from tainted code to API code.

6.3.2 AtomBombing

AtomBombing is a technique that injects a (malicious) code snippet into explorer.exe and executes it without using the APIs commonly used for code injection: WriteProcessMemory, VirtualAllocEx, and CreateRemoteThread. AtomBombing takes advantage of the atom table which is a shared data structure already mapped into explorer.exe. AtomBombing maps a code snippet into the virtual memory space of explorer.exe by writing it in the atom table. After the code mapping is completed, AtomBombing invokes the NtQueueApcThread API to hijack a thread with sleeping status in explorer.exe. Then, when the sleeping thread wakes up, the thread starts executing the mapped code. AtomBombing uses the return-oriented-programing (ROP) technique to avoid the issue in which the memory areas of the atom table do not have the executable permission.

6.3.2.1 Result

API Chaser successfully captured the API calls invoked from the code injected into explorer.exe since the caller instructions of these APIs were written by this synthetic malware, i.e., they were tainted. However, API Chaser missed capturing the API calls invoked from the ROP gadgets. The reason for this was that the caller instructions of these API calls in the ROP gadgets belonged to a benign code. So, these caller instructions have the benign-tag and control transfer of this API call is from benign-tag to api-tag. We discuss this issue in more depth in Section 8.3.3.

6.3.3 PowerLoaderEx

PowerLoaderEx is an evolved version of PowerLoader [6]. This is also a technique that injects a code snippet into a benign process and executes it without using the three APIs. PowerLoaderEx first writes code in a desktop heap memory, which is a shared area among GUI applications. Second, it overwrites the function pointer that handles a specific type of window message using the SetWindowLongPtr API. Then, it intentionally generates the window message to kick the handler with the SendNotifyMessage API. Since the desktop heap is basically not executable, PowerLoaderEx employs the ROP technique to add executable permission to the injected code in a manner similar to AtomBombing.

6.3.3.1 Result

Similar to the AtomBombing case, API Chaser could capture the API calls invoked from the injected code, while it failed to capture the ones from ROP gadgets. This is also discussed more in Section 8.3.3.

6.3.4 Shim-Based DLL Injection

Shim-based DLL Injection is a technique that injects a DLL into a benign process by taking advantage of the application compatibility mechanism of Windows; Microsoft officially prepares a mechanism that ensures backward compatibility in most of their products. This is currently implemented by the Application Compatibility Framework (ACF). The ACF is capable of intercepting API calls, controlling the loading process of DLLs, and patching memory. The PoC code for this technique [13] generates an sdb file while configuring its inject-target program, an injecting

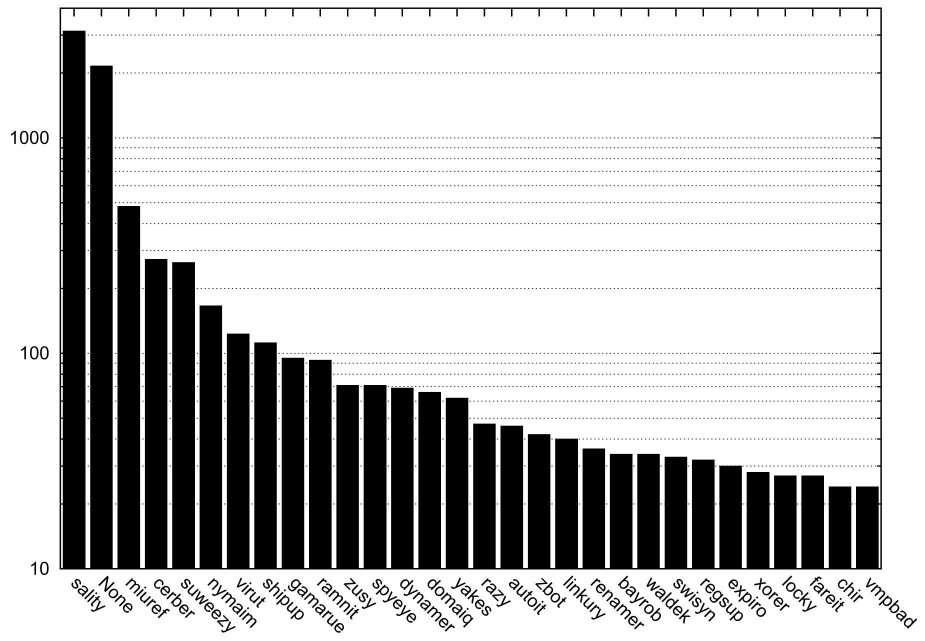


Fig. 8 Top 30 malware families in 6,722 samples. X-axis is malware family, while y-axis is the number of samples in each family with logarithmic scale.

DLL installs the sdb file with the `sdbinst` command, and the PoC code kicks the target program. When the target program is executed, the ACF injects the configured DLL into the process of the target program.

6.3.4.1 Result

API Chaser can track the injection of the DLL if the DLL is tainted. Taint analysis allows API Chaser to track the movement of its target code with taint tags without being affected by the injection manner. In real-world malware, since an injecting DLL is downloaded or dropped by the malware, the DLL becomes tainted on API Chaser. So, we do not miss capturing API calls invoked from the DLL injected into a process.

6.3.5 Stealth Loader

Stealth Loader is a program loader that loads Windows system DLLs such as kernel32.dll and ntdll.dll without leaving any trace to be detected. By loading a system DLL with Stealth Loader, the loaded DLL is not recognized as being 'loaded' by the Windows OS or even analysis tools. Since analysis tools fail to recognize the existence of the loaded DLL, they also fail to capture API calls of the functions exported from the unrecognized system DLL.

6.3.5.1 Result

API Chaser recognizes the calls of the function exported from stealth-loaded system DLLs as 'API calls'. This is because API Chaser identifies an API call based on taint tags added to the API before initiating an analysis (pre-boot disk tainting), and it does not rely on any metadata managed by the Windows OS, which is the portion Stealth Loader attacks. So, if Stealth Loader deceives the Windows OS by not leaving any trace identifying the existence of loaded DLLs, API Chaser is not affected by that.

6.4 Large-Scale Malware Analysis Experiment

The goal of this experiment is to show how much major hook and target evasion techniques are among real-world malware samples. To achieve this goal, we collected a certain number of malware from various sources and analyzed them with API Chaser to find malware using these techniques. We call malware using hook evasion techniques *hook-evasive* malware, whereas we call malware using target evasion techniques *target-evasive* one in this experiment.

6.4.1 Dataset

As the dataset for this experiment, we totally collected 8,979 malware samples from various data sources including malware exchange with an industrial vendor and our own honeypots. Then, we filtered out 100 samples whose hash value was duplicated. We used the 8,879 samples for this experiment. Next, we downloaded the anti-virus scan reports of 6,722 samples from VirusTotal [54] because we needed them for classification with AVClass [45]. The other 2,157 samples did not have any report in VirusTotal, which means they had never been uploaded for scanning onto VirusTotal, so we made None family for them. As a result, we classified the 6,722 samples into 420 families with AVClass and thus we classified 8,879 samples into 421 families including None.

Figure 8 shows the major top 30 malware families in the samples. Sality is the most major family in them and it has 3,131 samples, which occupies about 28% of the dataset. None is the second major family and it has 2,157 samples, which occupies about 24%. Since with only the two families, we can occupy about 60% of the total dataset. So, this dataset has a certain amount of bias. Thus, when we show the results of this experiment, we show not only the number of samples, but also the number of the families to mitigate this bias.

6.4.2 Procedure

As an analysis environment, we prepared three API Chaser instances to analyze the samples in parallel and we configured the analysis time to be 5 mins per analysis. After 5 mins elapsed, we forcibly terminated the API Chaser process even though the malware under analysis was still running. We used Win7sp1 as the guest OS of API Chaser for this experiment.

After all analyses had been done, we found malware samples using hook evasion or target evasion techniques by parsing each analysis log. We found hook-evasive malware samples from API call logs as follows. We first calculated the virtual addresses of each API from the base address of a loaded DLL and its export function table. Next, we compared the address logged in a log file as an API call destination to the calculated address, and then if the addresses were not matched, we identified the malware as hook-evasive.

We also found target-evasive malware samples as follows. We first created a list of processes which were in parent-child relationship with monitored malware based on specific API calls, such as CreateProcessA/W. When we found an API call invoked from a process which was not on the list, we identified the malware sample as target-evasive because the API call possibly came from a injected code with target evasion.

6.4.3 Result

We found 701 hook-evasive malware samples in the dataset. 476 samples out of 701 belonged to miuref malware family and this was the most popular one in the dataset. 104 samples belonged to None family and this is the second popular one. The others belonged to any one of 34 families. The top 5 families were miuref(476), None(104), ramnit(66), sality(3), and dynamer(3) whose numbers in the parentheses express the number of samples belonging to these families. Regarding target-evasive malware, we found 344 target-evasive malware samples. 56 samples out of 344 belonged to None family and this was the most popular one in the dataset, while the others belonged to any one of 83 families. The top 5 families were None(56), bayrob(32), sality(31), gamarue(21), and parite(19). Since hook evasion techniques were used in 8.5% families of all dataset families and target evasion techniques are used in 19.9%, we argue that these evasion techniques are major and often used among malware to intensionally hide API calls.

Through these analyses, we collected a total of 5,133,292,748 API call logs. 4,771,863 out of 5,133,292,748 API calls, which were about 0.09% of the total API calls, were intentionally hidden with hook evasion techniques, while 172,859,468 API calls, which were about 30% of the total API calls, came from a injected code with target evasion techniques. As we showed in the accuracy experiments, API Chaser was able to capture these API calls without being evaded and then we could collect the arguments passed to these API calls correctly, which possibly contain useful information as an indicator for detecting malware, i.e., indicator of compromise (IOC). However, if you analyze these evasive malware with an analysis environment whose architecture for API monitoring is based on the ones of Type I or Type II, you may miss capturing some of these API calls or excessively capture them, respectively, if the analysis environment does not care about evasion techniques at all. These inaccuracies of API call monitoring possibly lead to both false negatives and positives.

Lastly, regarding analysis times, we started the analysis at 2017/2/7 19:50:59 and finished at 2017/2/23 1:35:33. We spent approximately 382 hours to analyze the 8,879 samples. On average, we spent 7.75 minutes (= (382*60)/(8,879/3)) to analyze one sample with one API Chaser instance, even though we configured the analysis time to be set to 5 mins for the analysis. This time difference comes from the time for preparing the analysis environment before initializing the analysis and that for compressing logs after analysis.

6.5 Performance Experiment

The goals of this experiment are to show how much of performance degradation API Chaser has, compared to a vanilla QEMU, and where the degradation mainly comes from. Performance is an important factor to consider for a malware analysis system because a large delay in execution in a specific code block may expose the existence of an analysis environment to malware. To avoid being detected with the delay in execution, a practical malware analysis environment needs to achieve a reasonable level of performance. However, as we explained until here, we put a higher priority on precision for API monitoring rather than performance as the fundamental design of API Chaser. This design choice may impose on performance penalties. To know the impact of the penalties, we conduct this experiment and clarify the most influential part in API Chaser.

We used five Windows standard commands for this experiment. With these commands, we could cover APIs of major behaviors which we should focus on, such as file, registry, network, process, or memory-related behaviors.

As comparative environments, we prepared three environments: vanilla QEMU (`Qemu`), API Chaser without API monitoring (`w/o API monitoring`), and API Chaser without argument handlers (`w/o argument hander`). The reason why we prepared these comparative environments was to clarify which functionality of API Chaser mainly causes performance degradation. As we explained, we added several functionalities to QEMU to implement API Chaser; we could roughly classify the added functionalities into three groups: functionality related to taint analysis, API hooking (API monitoring), or argument handling. API Chaser, of course, has the three functionalities, i.e., taint analysis, API monitoring, and argument handling. The `w/o argument handler` environment has taint analysis and API monitoring but it does not have argument handing. The `w/o API monitoring` environment has only taint analysis and it does not have both API monitoring and argument handling. `Qemu` does not have any of them. By comparing the performances of them, we could identify the most influential functionality in the three ones to the performance of API Chaser. In addition, we used WinXPsp3 as the guest OS of API Chaser for this experiment.

6.5.1 Result

Figure 9 shows the relative run duration of these five commands on each environment compared to relative QEMU, which is set to 1. The results show that the degradation in performance of API Chaser was approximately 3 to 10 fold compared to that for `Qemu`. As you can see, the performance degradation mainly came from the taint analysis functionality because the difference between `Qemu` and `w/o API monitoring` was larger than the others, i.e., difference between `w/o argument handler` and `w/o API monitoring` or one between API Chaser and `w/o argument hander`.

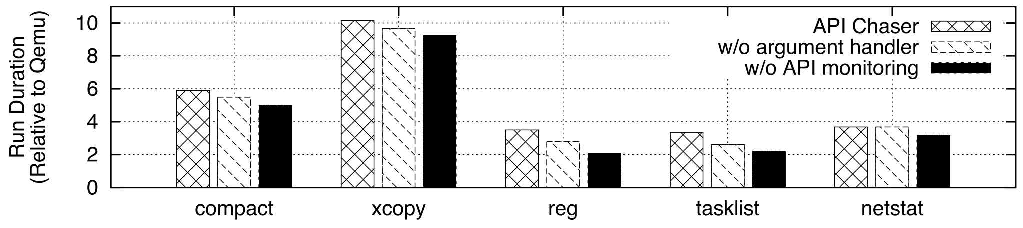


Fig. 9 Results of performance experiment: Number of captured API calls during the execution of each command is as follows: compact is 28,464, xcopy is 1,222, reg is 44,059, tasklist is 8,271, and netstat is 103.

We consider that the degradation is not a severe limitation to API Chaser because the current version of API Chaser has not been optimized to reduce its overhead. We consider that there is much room for improvement, for example, applying the work done in Ref. [18] to API Chaser. In addition, we discuss in Section 8.3.1 an issue that arises from the performance degradation when we analyze malware in terms of checking the delay of execution.

7. Related Work

In this section, we discuss several studies related to API Chaser. We categorize them into dynamic analysis sandbox, target evasion, and hook evasion.

7.1 Dynamic Analysis Sandbox

Several approaches have been proposed that precisely monitor malware activities based on API monitoring. We describe these approaches based on three categories: binary rewriting, binary-instrumentation, and simulation.

Binary rewriting approaches involve implanting hooks at the entries of APIs by modifying a function table for APIs or instructions of system DLLs in the analysis environment. CWSandbox [55] and Cuckoo Sandbox [38] employ an in-line hooking technique that replaces instructions at the entry of an API with a `jmp` instruction pointed to a function for monitoring. JoeBox [9] hooks APIs or system calls using a data rewriting technique, i.e., export address table hooking, that replaces a function pointer in the export address table of the PE header with the address to a function for monitoring.

Binary rewriting possibly exposes artifacts that allow malware to realize that it is running under analysis, and stop its execution or change its behavior. This drawback causes us to fail to grasp the actual activities of malware. We have not considered rewriting approaches in API Chaser because we want to avoid such exposure to malware.

Binary instrumentation involves comparing the address of instructions being executed with those where the API is located. Stealth Breakpoint [52] performs code instrumentation for user-land processes at the OS layer and determines the execution of the target address, i.e., the addresses where APIs reside, based on address comparison. Cobra [53] is a malware analysis environment that uses stealth breakpoints for hooking API calls. TTAnalyze [4] (ancestor of Anubis [3]) monitors APIs and system calls from malware in the virtual machine monitor (VMM) layer using address comparison. TTAnalyze determines target processes using Control Register number 3 (CR3), which is passed from a probe module running on the guest OS. Panorama [58] is a malware analysis environment established on the whole-system emulator TEMU [48]. Panorama is designed to analyze and detect malware based on taint tracking. It does not hook any APIs or system calls for malware analysis, although we found in its source code that TEMU has functions for hooking APIs based on address comparison.

These systems detect the execution of APIs by comparing the address pointed by an instruction pointer to addresses where APIs should reside. In addition, they identify the caller of an API based on PID, CR3 or TID. The anti-analysis techniques mentioned in Section 2 can possibly be used to evade these approaches. To address these evasion issues, we proposed the API monitoring mechanism with code tainting in API Chaser.

An approach that is similar to that proposed herein for monitoring API calls is the one applied in IntroLib [16] and CXPInspector [11]. Their approach defines an API call as a control transfer between different memory regions, i.e., one for malicious code and the other for API code. IntroLib relies on shadow page tables for interception, while CXPInspector relies on a hardware-assisted virtualization support feature such as Extended Page Tables (EPT) or Nested Page Tables (NPT). On the other hand, API Chaser relies on taint tags to intercept the control transfer between malicious code and API code. As we mentioned previously, taint analysis allows us to track the movement of both malicious codes and API codes. So, we can handle both hook evasion and target evasion in the same way.

Norman Sandbox [36] simulates the Windows OS and local area networks. It simulates almost all APIs that the Windows system library provides. However, it can also possibly be detected by malware because it does not perfectly simulate the behaviors of all Windows APIs.

7.2 Target Evasion

Another category of research related to this paper is code injection defense. Quincy [1] and Membrane [39] are designed to detect code injection for forensics use-cases. Quincy relies on a machine-learning approach with 38 features of code injection, while Membrane focuses on the anomalies caused by code loading. On the other hand, our primary use-case is dynamic analysis and we focus on tracking malicious code and capturing API calls invoked from injected malicious code. Bee Master [2] prepares decoy processes in an analysis environment and detects injections into processes. Bee Master primarily focuses on detecting code injection, while we mainly focus on analyzing the injected code. Korczynski et al. [28] proposed an approach for tracking malicious code injection with taint analysis, which is similar to code tainting. The difference is that we mainly focus on API monitoring and applying code tainting to API monitoring, i.e., not only for tracking injected code.

7.3 Hook Evasion

Yin et al. introduced HookFinder [57] to identify and analyze malware hooking behavior. Their study mainly focuses on detecting malware hook behavior and clarifying how they hook. On the other hand, we mainly focus on maintaining precise analysis even in a situation where malware adds hooks to API codes, i.e., stolen code. Sharif et al. proposed an approach for static analysis to analyze the control flow from a call site to the API code via junk instructions for static analysis [46], while our approach is mainly for dynamic analysis. Choi proposed an approach to track the movement of API codes with stolen code by tracing all memory access [14]. The proposed approach herein is similar to his, but the difference is that we track the movement with taint analysis, while he tracks using memory access trace.

8. Discussion

In this section, we discuss the resistance capability of API Chaser against hook evasion techniques which we did not have experiments in Section 6.2, multiple operands handing for taint propagation, and the limitations of API Chaser.

8.1 Other Hook Evasions

In the accuracy experiment in Section 6.2, API Chaser was not affected by hook evasion techniques used in real-world malware. However, we could not find any malware in the wild using name confusion or copied API obfuscation. So, we qualitatively discuss the resistance capability of API Chaser against these two techniques.

We consider that API Chaser is not affected by name confusion because of the following considerations. When a DLL is copied, the taint tags set on the DLL are propagated to the copied DLL. Even if the name of the DLL is changed, it does not affect taint propagation at all because taint propagation is conducted at (virtual) hardware layer without depending on the semantics of an OS or a file system and the change of file names is a matter of OS or file system layers. Thus, when API code in the copied DLL is executed, we can capture the execution of the API correctly because the propagated taint tag has existed on the code of the API.

We also consider that API Chaser is not affected by copied API obfuscation. As we explained in Section 2, copied API obfuscation is similar with stolen code. Copied API obfuscation copies all instructions of an API, while stolen code does the first few instructions of the API. Considering a case that a copied or stolen API is invoked from a malicious code, there is no difference in control transfers between them. That is, an execution control is transferred from the malicious code to the first instruction of the copied or stolen API. Since we have already demonstrated that API Chaser handles stolen code properly, we believe that it could do copied API obfuscation as well.

8.2 Multiple Operands for Taint Propagation

When API Chaser handles an instruction which has more than two operands and both of them have different taint tags, a taint propagation depending on one of the tags is disconnected. This is because, in our implementation of API Chaser, we simply propagate the taint tag of the first operand and discard the one of the second operand when we encounter this situation.

To mitigate this issue, an approach is to make a priority based on the types of taint tags for propagation. When we encounter this situation, we decide which tag should be propagated to the destination operand based on the priority. Another one is to generate a new tag and make a relationship between the new tag and the tags of each operand to construct the data-flow with them in an offline analysis. In either case, we need to add a code to the handlers for these types of instructions, i.e., instructions with multiple operands. Since these instructions often appear during an analysis, the impact of the additional code is not so small. We need to achieve a balance between performance and precision when we adopt one of these approaches to API Chaser.

8.3 Limitation

We discuss limitations of API Chaser from viewpoints of detection-type anti-analyses, scripts, return-oriented-programming, and implicit flow.

8.3.1 Detection-Type Anti-analysis

With the exception of evasion-type anti-analysis, malware often uses detection-type anti-analysis techniques [56]. Regarding this type of anti-analysis, API Chaser performs well except for VM detection and timing attacks because API Chaser does not modify the guest OS environment, does not install any modules, and does not simulate any APIs. So, we discuss the two exceptions below.

Several methods for detecting QEMU have been studied and proposed [19], [32], [33], [42]. To avoid these detections, we individually managed to let QEMU-specific artifacts become invisible to malware. For example, we changed the product names of virtual hardware in QEMU for detection techniques that depends on these names. We also changed the behaviors of specific instructions by finding the execution of these instructions and dynamically patching them at runtime.

Timing attacks are a technique that checks the delay in executing a specific code block. We designed API Chaser to focus on accuracy rather than performance; therefore, it takes several more seconds to execute part of a code block than in real hardware environments. As for this technique, we can overcome this with the same approach as that used in our previous study [24], which controls the clock in a guest OS on API Chaser by adjusting the tick counts in the emulator to remove the delay.

8.3.2 Scripts

API Chaser has a limitation for analyzing script-type malware, e.g., a visual basic script or a command script. These scripts are executed on some platforms such as an interpreter or a virtual machine. Although these scripts have the taint tags of malware, API Chaser cannot detect their execution because the instructions executed on the virtual CPU are those of their platform and not those of the tainted script. To address this problem, we are currently considering a way to identify the target code with both taint tags and semantic information such as PID and TID.

8.3.3 Return Oriented Programming

API Chaser faces a limitation when an API is called in a manner similar to ROP [12]. For example, when an attacker constructs a ROP chain by pushing the address of a specific API and executing the `ret` instruction in a benign code region to jump to the API, the control transfer in this case is from a benign-tag to an api-tag. So, API Chaser fails to identify this control transfer as a monitoring target.

We may be able to handle this case by extending taint-based control transfer interception using the approach that Korczynski et al. [28] proposed. That is, when a control transfer instruction such as `ret`, `call`, or `jmp` is executed and the destination address of the instruction is tainted, we identify such a control transfer and the first basic block at the destination address as a part of malicious code.

Another option is to detect the ROP code. If we can detect the ROP code, we may be able to identify the execution of APIs called from malware via the detected ROP code. Detection of ROP is outside the scope of this paper and we leave it for other studies. Many studies leverage the unique behavioral characteristics of the ROP code such as its use of many ret instructions, jumps to the middle of an API, or jumps to an instruction of non-exported functions.

8.3.4 Implicit Flow

Another limitation of API Chaser is due to feasibility issues of taint propagation, e.g., implicit flow. If malware authors know the internal architecture of API Chaser, especially code taint propagation, it may be possible to cause intentionally API Chaser to yield false positives or false negatives using implicit flow. For example, malware reads a piece of code in a benign program and processes the code through implicit flow which does not change its value. Then it writes the code back to the same position. As a result, the taint tags on the code are changed from benign to malware. Due to this, if malware executes the written code, API Chaser identifies the execution as the one of malware, even though the code is truly equivalent to benign code. On the other hand, if malware reads a piece of code in an API and conducts the same process, it overwrites the taint tags for API with those for malware. Thus, API Chaser deals with the execution of the code as one of malware. To address this problem, we must improve the strength of the taint propagation, for example, as done in Refs. [22], [47]. We consider this as our future work.

9. Conclusion

The anti-analysis feature of malware is a challenging problem for anti-malware research especially in developing practical malware analysis environments. We focused on this problem and provided a solution by using API Chaser, which is a prototype system of our API monitoring approach. API Chaser was designed and implemented to prevent malware from evading API monitoring. We conducted experiments using actual malicious code with various types of anti-analysis to show that API Chaser correctly works according to its design of being difficult to evade. We believe that API Chaser will be able to assist malware analysts in understanding malware activities more correctly without spending a large amount of effort in reverse engineering and contribute to improving the effectiveness of anti-malware research based on API monitoring.

References

[1] Barabosch, T., Bergmann, N., Dombeck, A. and Padilla, E.: Quincy: Detecting Host-Based Code Injection Attacks in Memory Dumps, *Detection of Intrusions and Malware, and Vulnerability Assessment*, Polychronakis, M. and Meier, M. (Eds.), Cham, Springer International Publishing, pp.209–229 (2017).

[2] Barabosch, T., Eschweiler, S. and Gerhards-Padilla, E.: Bee Master: Detecting Host-Based Code Injection Attacks, *Detection of Intrusions and Malware, and Vulnerability Assessment*, Dietrich, S. (Ed.), Cham, Springer International Publishing, pp.235–254 (2014).

[3] Bayer, U., Habibi, I., Balzarotti, D., Kirda, E. and Kruegel, C.: A View on Current Malware Behaviors, *Proc. 2nd USENIX Conference on Large-scale Exploits and Emergent Threats: Botnets, Spyware, Worms, and More*, *LEET'09*, p.8, USENIX Association (online), available from ⟨http://dl.acm.org/citation.cfm?id=1855676.1855684⟩ (2009).

[4] Bayer, U., Kruegel, C. and Kirda, E.: TTAnalyze: A Tool for Analyzing Malware, *Proc. European Institute for Computer Antivirus Research Annual Conference* (2006).

[5] Bellard, F.: QEMU, a Fast and Portable Dynamic Translator, *USENIX Annual Technical Conference, FREENIX Track*, pp.41–46, USENIX (2005).

[6] BreakingMalware.com: PowerLoaderEx, enSilo (online), available from ⟨https://github.com/BreakingMalware/PowerLoaderEx⟩ (accessed 2018-10-01).

[7] BreakingMalwareResearch: atom-bombing, enSilo (online), available from ⟨https://github.com/BreakingMalwareResearch/atom-bombing⟩ (accessed 2018-10-01).

[8] Brumley, D., Hartwig, C., Liang, Z., Newsome, J., Song, D.X. and Yin, H.: Automatically Identifying Trigger-based Behavior in Malware, *Botnet Detection*, pp.65–88 (2008).

[9] Buhlmann, S.: Joebox Sandbox, Joe Security LLC (online), available from ⟨http://www.joesecurity.org/⟩ (accessed 2018-10-01).

[10] Carrier, B.: The Sleuth Kit (TSK), http://www.sleuthkit.org/ (online), available from ⟨http://www.sleuthkit.org/⟩ (accessed 2017-08-17).

[11] Carsten Willems, Ralf Hund, T.H.: CXPInspector: Hypervisor-Based, Hardware-Assisted System Monitoring, Technical Report Technical Report TR-HGI-2012-002, Ruhr University Bochum (2012).

[12] Chen, P., Xiao, H., Shen, X., Yin, X., Mao, B. and Xie, L.: DROP: Detecting Return-Oriented Programming Malicious Code, *Proc. 5th International Conference on Information Systems Security, ICISS '09*, pp.163–177, Springer-Verlag (2009).

[13] Chevet, S.: dllinjshim.cpp, github.com (online), available from ⟨https://gist.github.com/w4kfu/95a87764db7029e03f09d78f7273c4f4⟩ (accessed 2018-10-01).

[14] Choi, S.: API Deobfuscator: Identifying Runtime-obfuscated API calls via Memory Access Analysis, Black Hat Asia (2015).

[15] Coppola, M.: Loadvm snapshot as read-only, Qemu Mailing List (online), available from ⟨https://bugs.launchpad.net/qemu/+bug/1184089⟩ (accessed 2018-10-01).

[16] Deng, Z., Xu, D., Zhang, X. and Jiang, X.: IntroLib: Efficient and transparent library call introspection for malware forensics, *Digital Investigation*, Vol.9, pp.S13–S23 (online), DOI: https://doi.org/10.1016/j.diin.2012.05.013 (2012).

[17] Egele, M., Scholte, T., Kirda, E. and Kruegel, C.: A Survey on Automated Dynamic Malware-analysis Techniques and Tools, *ACM Comput. Surv.*, Vol.44, No.2, pp.6:1–6:42 (online), DOI: 10.1145/2089125.2089126 (2008).

[18] Ermolinskiy, A., Katti, S., Shenker, S., Fowler, L.L. and McCauley, M.: Towards Practical Taint Tracking, Technical Report UCB/EECS-2010-92, EECS Department, University of California, Berkeley (2010).

[19] Ferrie, P.: Attacks on Virtual Machine Emulators, Symantec Security Response (2006).

[20] Hex-Rays: Hex-Rays Home, Hex-Rays (online), available from ⟨https://www.hex-rays.com/⟩ (accessed 2018-09-30).

[21] Iwamura, M., Itoh, M. and Muraoka, Y.: Towards Efficient Analysis for Malware in the Wild, *Proc. IEEE International Conference on Communications, ICC '11* (2011).

[22] Kang, M.G., McCamant, S., Poosankam, P. and Song, D.: DTA++:

Dynamic Taint Analysis with Targeted Control-Flow Propagation, *NDSS* (2011).

[23] Kawakoya, Y., Iwamura, M. and Hariu, T.: Tracing Malicious Code with Taint Propagation, *Journal of Information Processing Society of Japan*, Vol.54, No.8, pp.2079–2089 (2013).

[24] Kawakoya, Y., Iwamura, M. and Itoh, M.: Memory Behavior-Based Automatic Malware Unpacking in Stealth Debugging Environment, *Proc. 5th IEEE International Conference on Malicious and Unwanted Software* (2010).

[25] Kawakoya, Y., Iwamura, M., Shioji, E. and Hariu, T.: API Chaser: Anti-analysis Resistant Malware Analyzer, *Research in Attacks, Proc. Intrusions, and Defenses: 16th International Symposium, RAID 2013*, pp.123–143 (2013).

[26] Kawakoya, Y., Shioji, E., Otsuki, Y., Iwamura, M. and Yada, T.: Stealth Loader: Trace-free Program Loading for API Obfuscation, *Research in Attacks, Proc. Intrusions, and Defenses: 20th International Symposium, RAID 2017* (2017).

[27] 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, CCS '11*, pp.285–296, ACM (online), DOI: 10.1145/2046707.2046740 (2011).

[28] Korczynski, D. and Yin, H.: Capturing Malware Propagations with Code Injections and Code-Reuse Attacks, *Proc. 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS '17*, pp.1691–1708, ACM (online), DOI: 10.1145/3133956.3134099 (2017).

[29] Leitch, J.: Process Hollowing, AutoSec Tools (online), available from ⟨https://www.autosectools.com/Process-Hollowing.html⟩ (accessed 2018-10-01).

[30] Liberman, T.: ATOMBOMBING: BRAND NEW CODE INJECTION FOR WINDOWS, enSilo (online), available from ⟨https://blog.ensilo.com/atombombing-brand-new-code-injection-for-windows⟩ (accessed 2018-03-30).

[31] m0n0ph1: Process-Hollowing, autosectools.com (online), available from ⟨https://github.com/m0n0ph1/Process-Hollowing⟩ (accessed 2018-10-01).

[32] Martignoni, L., Paleari, R., Fresi Roglia, G. and Bruschi, D.: Testing System Virtual Machines, *Proc. 19th International Symposium on Software Testing and Analysis, ISSTA '10*, pp.171–182, ACM (online), DOI: 10.1145/1831708.1831730 (2010).

[33] Martignoni, L., Paleari, R., Roglia, G.F. and Bruschi, D.: Testing CPU Emulators, *Proc. 18th International Symposium on Software Testing and Analysis, ISSTA '09*, pp.261–272, ACM (online), DOI: 10.1145/1572272.1572303 (2009).

[34] McLoughlin, M.: The QCOW2 Image Format, people.gnome.org (online), available from ⟨https://people.gnome.org/ markmc/qcow-image-format.html⟩ (accessed 2018-10-01).

[35] Newsome, J. and Song, D.: Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software (2005).

[36] Norman: Norman Sandbox Analyzer, Norman.com (online), available from ⟨http://download01.norman.no/product_sheets/eng/SandBox_analyzer.pdf⟩ (accessed 2018-10-01).

[37] Nowak, T. and Sawicki, A.: The Undocumented Functions, undocumented.ntinternals.net (online), available from ⟨http://undocumented.ntinternals.net/⟩ (accessed 2018-03-13).

[38] Oktavianto, D. and Muhardianto, I.: *Cuckoo Malware Analysis*, Packt Publishing (2013).

[39] Pék, G., Lázár, Z., Várnagy, Z., Félegyházi, M. and Buttyán, L.: Membrane: A Posteriori Detection of Malicious Code Loading by Memory Paging Analysis, *Computer Security – ESORICS 2016*, Askoxylakis, I., Ioannidis, S., Katsikas, S. and Meadows, C. (Eds.), Cham, Springer International Publishing, pp.199–216 (2016).

[40] Pierce, S.: Defending Against Malicious Application Compatibility Shims, Black Hat Europe Briefings (2015).

[41] Portokalidis, G., Slowinska, A. and Bos, H.: Argos: An emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation, *Proc. 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006, EuroSys '06*, pp.15–27, ACM (2006).

[42] Raffetseder, T., Kruegel, C. and Kirda, E.: Detecting System Emulators, *Information Security*, Garay, J.A., Lenstra, A.K., Mambo, M. and Peralta, R. (Eds.), Berlin, Heidelberg, Springer Berlin Heidelberg, pp.1–18 (2007).

[43] React OS Project: ReactOS, React OS Project (online), available from ⟨http://www.reactos.org/⟩ (accessed 2012-12-13).

[44] Sathyanarayan, V.S., Kohli, P. and Bruhadeshwar, B.: Signature Generation and Detection of Malware Families, *ACISP*, pp.336–349 (2008).

[45] Sebastián, M., Rivera, R., Kotzias, P. and Caballero, J.: AVclass: A Tool for Massive Malware Labeling, *Research in Attacks, Intrusions, and Defenses*, Monrose, F., Dacier, M., Blanc, G. and Garcia-Alfaro, J. (Eds.), Cham, Springer International Publishing, pp.230–253 (2016).

[46] Sharif, M., Yegneswaran, V., Saidi, H., Porras, P. and Lee, W.: Eureka: A Framework for Enabling Static Malware Analysis, *Computer Security - ESORICS 2008*, Jajodia, S. and Lopez, J. (Eds.), Berlin, Heidelberg, Springer Berlin Heidelberg, pp.481–500 (2008).

[47] Slowinska, A. and Bos, H.: Pointless tainting?: Evaluating the practicality of pointer tainting, *Proc. 4th ACM European Conference on Computer Systems, EuroSys '09*, pp.61–74, ACM (2009).

[48] Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P. and Saxena, P.: BitBlaze: A New Approach to Computer Security via Binary Analysis, *Proc. 4th International Conference on Information Systems Security, ICISS '08*, pp.1–25, Springer-Verlag (2008).

[49] Suenaga, M.: A Museum of API Obfuscation on Win32, Symantec Security Response (2009).

[50] The Volatility Framework: The Volatility Foundation, The Volatility Foundation (online), available from ⟨https://www.volatilityfoundation.org/⟩ (accessed 2018-10-01).

[51] Themida: Temida — Overview, Oreans Technologies (online), available from ⟨http://www.oreans.com/themida.php⟩ (accessed 2018-10-01).

[52] Vasudevan, A. and Yerraballi, R.: Stealth breakpoints, *21st Annual Computer Security Applications Conference*, pp.381–392 (2005).

[53] Vasudevan, A. and Yerraballi, R.: Cobra: Fine-grained Malware Analysis using Stealth Localized-Executions, *Proc. 2006 IEEE Symposium on Security and Privacy, Oakland'06* (2006).

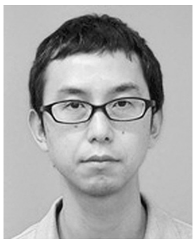
[54] VirusTotal: VirusTotal, virustotal.com (online), available from ⟨https://www.virustotal.com/⟩ (accessed 2018-09-28).

[55] Willems, C., Holz, T. and Freiling, F.: Toward Automated Dynamic Malware Analysis Using CWSandbox, *IEEE Security and Privacy*, Vol.5, No.2, pp.32–39 (online), DOI: 10.1109/MSP.2007.45 (2007).

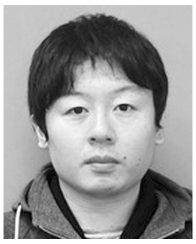
[56] Yason, M.V.: The Art of Unpacking, Black Hat USA Briefings (2007).

[57] Yin, H., Liang, Z. and Song, D.: HookFinder: Identifying and Understanding Malware Hooking Behaviors, *Proc. Network and Distributed System Security Symposium, NDSS 2008* (2008).

[58] Yin, H., Song, D., Egele, M., Kruegel, C. and Kirda, E.: Panorama: Capturing system-wide information flow for malware detection and analysis, *Proc. 14th ACM Conference on Computer and Communications Security, CCS '07*, pp.116–127, ACM (2007).



Yuhei Kawakoya received his B.E. and M.S. degrees in science and engineering from Waseda University in 2003 and 2005, respectively. He has been engaged in R&D since 2005 on computer security. From 2013 to 2016, he was engaged in R&D at NTT Innovation Institute, Inc. as a software engineer. He is a member of IPSJ and IEICE.



Eitaro Shioji received his B.E. degree in computer science and M.E. degree in communications and integrated systems from Tokyo Institute of Technology in 2008 and 2010, respectively. Since joining NTT in 2010, he has been engaged in R&D on computer security. He is a member of IEICE.

Makoto Iwamura received his B.E., M.E., and D.Eng. degrees in science and engineering from Waseda University, Tokyo, in 2000, 2002, and 2012, respectively. He joined NTT in 2002. He is currently with NTT Secure Platform Laboratories where he is engaged in the Cyber Security Project. His research interests include reverse engineering, vulnerability discovery, and malware analysis.

Jun Miyoshi received his B.E. and M.E. degrees in system science from Kyoto University in 1993 and 1995, respectively. Since joining NTT in 1995, he has been researching and developing network security technologies. From 2011 to 2016, he was engaged in R&D strategy management at NTT Secure Platform Laboratories. Now he is a research group leader of the Cyber Security Project in the NTT Secure Platform Laboratories. He is a member of IEICE.