Hindawi Publishing Corporation
Mathematical Problems in Engineering
Volume 2013, Article ID 402438, 11 pages
http://dx.doi.org/10.1155/2013/402438

Research Article

Osiris: A Malware Behavior Capturing System Implemented at Virtual Machine Monitor Layer

Ying Cao,[1] Qiguang Miao,[1] Jiachen Liu,[1] and Weisheng Li[2]

[1] *School of Computer Science and Technology, Xidian University, P.O. Box 167, Xi'an, Shaanxi 710071, China*
[2] *College of Computer Science and Technology, Chongqing University of Posts and Telecommunications, Chongqing, China*

Correspondence should be addressed to Qiguang Miao; qgmiao@mail.xidian.edu.cn

Received 17 January 2013; Accepted 19 March 2013

Academic Editor: Yuping Wang

Copyright © 2013 Ying Cao et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

To perform behavior based malware analysis, behavior capturing is an important prerequisite. In this paper, we present Osiris system which is a tool to capture behaviors of executable files in Windows system. It collects API calls invoked not only by main process of the analysis file, but also API calls invoked by child processes which are created by main process, injected processes if process injection happens, and service processes if the main process creates services. By modifying the source code of Qemu, Osiris is implemented at the virtual machine monitor layer and has the following advantages. First, it does not rewrite the binary code of analysis file or interfere with its normal execution, so that behavior data are obtained more stealthily and transparently. Second, it employs a multi-virtual machine framework to simulate the network environment for malware analysis, so that network behaviors of a malware are stimulated to a large extend. Third, besides network environment, it also simulates most common host events to stimulate potential malicious behaviors of a malware. The experimental results show that Osiris automates the malware analysis process and provides good behavior data for the following detection algorithm.

1. Introduction

Malicious software (or malware) is one of the most serious and invariant security threats facing the information system today, leading to a constant competition between malware authors and security analysts as technology evolves. Traditional static analysis ways look for context-based characteristic byte sequences to detect a malware. It requires highly experienced analysts to analyze source or assembly codes of a malware by using debuggers or disassembly tools, which is a laborious manual process. To evade this kind of detection, code obfuscations, including polymorphism and metamorphism [1], are then employed by malware writers. Polymorphic malwares constantly change in a variety of ways such as filename changes, compression, and encryption with variable keys. Metamorphic malwares even attempt to obfuscate the entire codes from infection to infection. Besides, another obvious limitation of traditional static ways is that they cannot handle unknown malwares. This is determined by their design principles. These limitations greatly threaten the effectiveness of classical static analysis methods [2].

To compensate these limitations, dynamic behavior-based analysis [3, 4] is proposed, which partly circumvents the problem of code obfuscation and automates the analysis process. A standard behavior based malware analysis process consists of three parts, including capturing program behavior, extracting behavioral features, and detection algorithm design. Among them, behavior capturing is an important prerequisite. It can be said that without accurate behavior data, no detection algorithm could give a correct result. In this paper, we focus on the problem of capturing program behavior.

On Windows platform, operating system provides (Application Programming Interface) API calls for programs on different layer of abstraction to perform common tasks, such as creating files, modifying registry keys, and establishing a network connection. They represent the interactions between programs and operating system, and therefore are the first choice of behavior data. To capture API

calls of a specific process, hooking and debugging are two prevalent ways. However, they suffer obvious weaknesses. First and foremost, hook, or debugging may interfere with the normal execution of the process under analysis, because they always change the execution path of analysis program, which probably leads to execution crash, making the analysis program terminated abnormally. Second, these techniques may be easy to detect and thus to circumvent. Through integrality checking, a malware can easily detect whether it is being monitored by the debugger; after that it can take some measures to escape monitoring [5]. Third, to do better behavior analysis, we need not only API calls invoked by the main process, but also API calls invoked by the child processes, the injected processes once process injection happens, and the service processes if main process creates them. It is not an easy and direct task for hooking and debugging ways to monitor all these processes automatically while main process is running. In a conclusion, they are not sophisticated enough for behavior based malware analysis.

Since malwares potentially do harm to computers in the network, when analyzing them, we do not directly run them in a real computer, but in a controlled virtual execution environment. By the rapid development of virtualization technology, virtual machine and system emulator become the best choices in constructing malware analysis environment. However, a controlled environment is the first prerequisite for security consideration, but is not enough for behavior based analysis. If the analysis environment totally disconnects with the Internet, network activities of analysis program will be sure to fail. This yields undesired results [6]. Besides, whether a malware performs malicious behavior may also depend on a certain host event, such as removable storage plugging in or off, network connection, or mouse clicking, or it has hidden behavior. In these cases, more simulation work must be done, especially the network environment simulation.

In this paper, we study the problem of behavior capturing for dynamic analysis. Our objective is to design an automatic program behavior capturing system to overcome limitations described above, from API call interception to how to construct virtual execution environment. A good behavior capturing system should have a sophisticated strategy to monitor not only behavior of main process of the analysis program, but also behavior of all newly created processes while the main process is running. The behavior capturing should be more stable and effective, making the analysis program hardly escape monitoring. Moreover, to get more accurate behavior data, complete simulation work should be done, including host events simulation and network environment simulation. For these ends, we present Osiris, a system implemented at virtual machine monitor layer to capture program behavior.

2. Related Work

In Windows system, API calls are frequently used to specify program behavior, because they are good representations that abstract richer semantics from implementation details. To intercept API calls of a specific process, the hook function is a most common way. If source code of analysis program is available, invocation to the hook function can be inserted into the source code at appropriate places, otherwise, binary rewriting is used. Additionally, there are two choices to implement binary rewriting. The first one is rewriting the original function. Particularly, first few instructions of the original function are rewritten so that prior to executing its original codes, hook function is invoked first. The second one is to rewrite all function sites so that the hook function is invoked instead of the original function. To this end, Detour [7] is a readily available library to implement function hooking on Windows platform. It provides not only convenient ways to modify binary files before they are loaded, but also ways to manipulate the memory images after a binary file is loaded. Besides, using debugging technique, breakpoints can be inserted at either call sites or the function body, thus an instrumented debugger can also be used to capture API calls.

In the research of building environment for malware analysis, virtual machines are often used to construct honey pot or honey net [8]; because they provide a well isolated execution environment and a full control of the entire computer system, a rich source of information can be obtained, ranging from hardware status to data in memory. However, commercial virtual machine usually does not provide second-development interfaces to directly retrieve information needed. This brings inconvenience to security researches. Therefore, open source virtual machines and system emulators are preferred to be used by many researchers. The most representative one is Qemu [9]. TTAnalyze is just a Qemu-based system to analyze unknown binaries [10], and later, it evolves into Anubis. The analysis is performed by monitoring the invocation of Windows API calls; besides, their parameters are also examined and tracked. It monitors API calls of analysis program through a method of comparing address of virtual instruction pointer with API entry point address. Finally, an expressive report is generated. CFISandbox [11] is another online malware analysis tool provided by Sunbelt. It employs a new hooking mechanism by synthesizing many existing binary rewriting techniques. A monitoring function that can perform analysis is installed before an API call is invoked and after it returns. It records both API and system calls. The output of an analysis run is a report file from a high semantic level. Norman Sandbox [12] is a dynamic malware analysis tool which focuses on detecting malwares that spread and replicate via email, P2P, or networks shares, especially worms. It executes analysis file in a tightly controlled virtual environment. All network traffics are redirected to simulated components and there is only one exception, that is, file downloading. The file under analysis is allowed to download files from real Internet, as the author claims that it is not a security problem to feed additional information into the simulated system. However, the downloading may fail for an out of date network location. In Norman Sandbox traditional hook is used to intercept API calls. The observed behavior is then written to a log file.

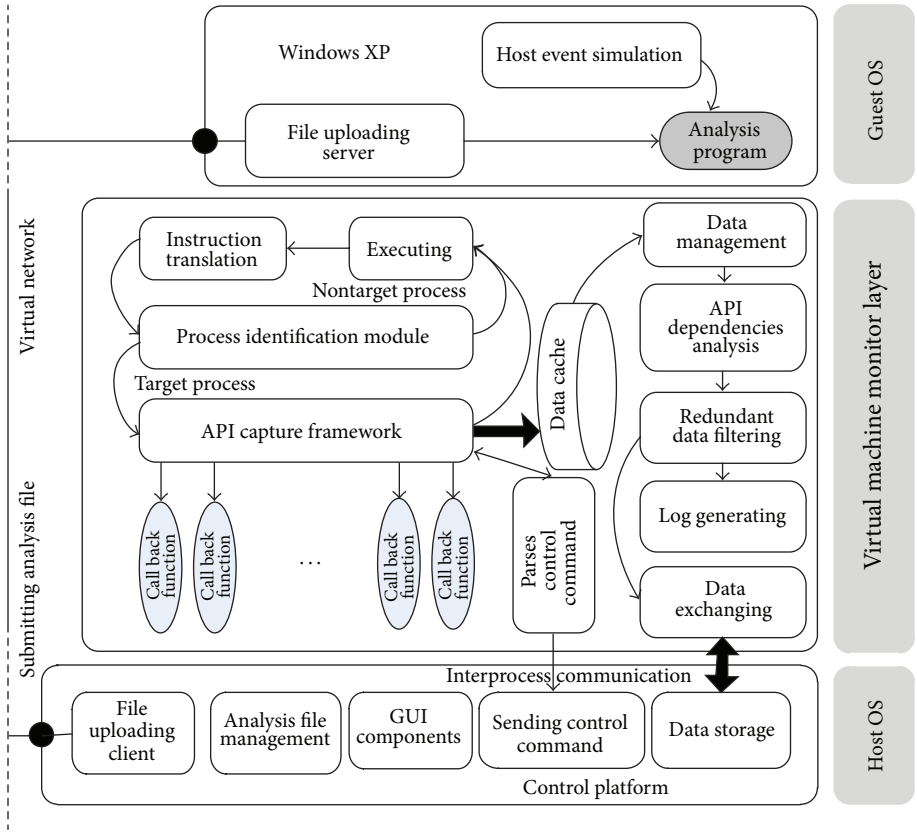


Figure 1: System overview of Osiris.

3. System Overview

In this section we give system overview of Osiris. The objective of Osiris is to capture behavior of suspicious executable files in Windows system. To this end, it monitors a collection of user mode and kernel mode API calls, as well as their arguments. Meanwhile, it extracts module information and security relevant OS kernel data directly from the virtual memory. Moreover, it also automatically simulates various host events to stimulate the hidden behavior of a malware as possible. To monitor network activities, Osiris employs a multivirtual machine framework to completely simulate the network environment. On PC platform, a most common setup includes Intel x86 architecture and Microsoft Windows operating system. So we choose this combination to implement the prototype of Osiris consisting of four parts: a modified Qemu which acts as emulator component, an ROS (http://www.mikrotik.com/software.html) which is used to redirect network traffic, a Qemu in which almost all prevalent network services are simulated, and a control platform which manages the whole analysis process and interacts with users. Figure 1 is the system overview of core functionality module of Osiris. The entire network deployment can be found and further introduced in Figure 3 in Section 5. The modified Qemu is the core component of Osiris. We start it from a snapshot of every analysis, for insurance of a clean analysis environment. Besides, guest OS and the local host are connected via a VLAN, in order that the control platform can upload the analysis program into guest OS and control its execution.

(1) API monitoring: to overcome the deficiencies that are shared by traditional hooking or debugging ways described in Section 1, Osiris monitors API calls of analysis program without modifying its binary code. The analysis program is running in a virtualized guest OS, and the behavior capture is implemented inside Qemu, namely, the virtual machine monitor, which is a more privileged layer than guest OS, so that the analysis programs can hardly escape monitoring. The behavior of any newly created process while main process is running is monitored, including child processes, injected processes, and service processes, if exists.

(2) Network simulation: to analyze network behavior of a malware, almost all kinds of online services in Internet should be simulated within a constrained environment, including DNS, FTP, HTTP, and email service, we implement this with the help of ROS which redirects all network traffic to another Qemu in which network services are simulated. Instead of simply installing the necessary service programs,

bogus responses are also constructed to stimulate network behavior of a program to a large extent.

(3) Host events simulation: in guest OS, Osiris simulates the most common host events automatically to stimulate potential hidden behavior of a malware as possible, including removable storage plug-in/plug-out, cdrom plug-in/plug-out, mouse clicking, microphone turning on/turning off, camera turning on/turning off, network shared folder connection, and time flowing.

(4) A second time execution: for many Trojans, it is common that they release a file and set it to automatically start next time the computer boots and real malicious behavior is in the released files. There are also malwares which register a service to operating system and the service will not start until next time the computer boots. In these cases, a single execution of analysis file gets little useful information. To get accurate behavior data, Osiris automatically executes the released files or starts the services, which will not be started until next time computer boots, and after that monitors their behavior.

In the following sections, we go into more design details of Osiris.

4. Design Details

4.1. Identifying the Main Process. Recall that in Osiris, behavior capturing is implemented at the virtual machine monitor layer to overcome limitations shared by traditional hooking or debugging ways. However, the semantic gap between virtual machine monitor and guest operating system is the very first problem that is needed to solve. An unmodified virtual machine or emulator dutifully emulates hardware of the computer system. It does not need to know process, which is a concept at operating system layer. However, malware analysis requires behavior data of certain target processes. Behavior of other programs in operating system is redundant and helpless for analysis. Therefore, to perform behavior capturing, we must teach the virtual machine monitor to know what process is at first. Besides, executing analysis file in the virtual execution environment, main process may create child processes, inject into system processes, or create services. These newly created processes should be identified automatically while the main process is executing. They make up the entire monitoring targets. We call this problem process identification.

From the perspective of process manager in operating system, a process is made up of kernel data structures and the virtual memory address space. In Windows system, (Page Directory Base) PDB address is used to identify the unique virtual memory space allocated to a process. Once a process is scheduled to start executing, operating system stores its PDB address into CR3 register. Thus a possible way to identify main process at the virtual machine monitor layer can be concluded as follows: first, get PDB address of the target program during process initialization and then by comparing it with the value of CR3 register, whether the target program is executing or not can be decided.

The prerequisite of the above methods is getting PDB address of the main process. Windows maintains a kernel data structure called EPROCESS for every active process to accomplish process management. PDB address is just a member of EPROCESS. Since EPROCESS lies in the kernel space, it is natural to write a kernel mode program to get its value. However, such method has disadvantages. Firstly, the implementation of a kernel mode program is complicated. Secondly, a program to get EPROCESS value of the target program must run in guest OS; therefore, extra network communications are needed for data exchange. Thirdly, according to recent research, it may be unreliable to get data directly through interface provided by guest OS, because they could be tampered. It is suggested that in a virtual machine system, data directly obtained at the most privileged layer, virtual machine monitor layer, is the best choice [13]. Therefore, we design a method to read PDB address and any other data in kernel space by directly searching virtual memory, which solves each of the three problems above.

According to the analysis above, the key step of identifying the main process is to get value of its EPROCESS. Windows system maintains a data structure called KPCR (Kernel Processor Control Region) for each CPU in order to keep some global information for thread switching. KPCR is located at 0xFFDFF000 in the linear address space, and inside it, there exists another data structure called KPRCB (Kernel Processor Control Block) at 0xFFDFF120. The locations of KPCR and KPRCB typically do not change with the update of Windows, so they are stable. Additionally, there is a data structure called KTHREAD, lying in the location that is 0x04 bytes from the starting address of KPRCB, and EPORCESS is just 0x44 bytes from the starting address of KTHREAD. Now that we get where EPROCESS lies in the memory, then we can read its value. Furthermore, EPROCESSs of all the active processes are organized in a double-linked list. If we get EPROCESS of any running process, by traversing this double-linked list, EPROCESS of the target process wanted can be found; with this in hand, almost all of the important information of a process is available, such as PDB value and process name, which are 0x18 bytes and 0x174 bytes from the starting address of EPROCESS, respectively.

In summary, our method to identify the main process at the virtual machine monitor layer is as follows.

(1) Before the execution of each translation block, use memory access functions provided by Qemu to read process name and PDB value of current active process from EPROCESS in kernel space.

(2) Decide whether the process name is the same with target process name. If so, store PDB value and call it T.

(3) Monitor CR3 register and compare its value to T; every time they are equal, it means the target process is running; then start behavior data capturing procedure.

4.2. Capturing API Calls and Their Arguments. After identifying main process, the next step is to monitor its behavior by intercepting API calls invoked by it. From disassembly instructions, it is clear to see that calling an API call follows a same pattern, that is, using several push instructions to push calling arguments to function stack. Then use a call instruction to change the execution flow to the entry point of this API. In instruction emulation, Qemu uses translation blocks and every translation block is ended up with a jump kind instruction, including direct jump, such as jmp, call, and ret, and conditional jump, such as jz, jc, and je [14]. It can be concluded that in Qemu, the first instruction of an API call must be the first instruction of a certain translation block. Thus an API call can be detected by comparing its entry point address to instruction pointer of current translation block. Based on this principle, we then design the API capturing module. It is a callback function system and is also the core functionality module in Osiris. In the implementation, every monitored API has two corresponding callback functions. Once an API is called, the callback functions will get back its parameters from the virtual memory. To implement this callback system, modifications must be made to the translation process of Qemu, so that corresponding callback function can be invoked before the execution of the API.

The next question is how to get the entry point address of a particular API. In Windows, API calls are exported from DLL (Dynamic Link Library) and entry addresses of API calls can be found in the exporting table of DLLs. In Osiris, 187 application layer API calls and 73 kernel API calls are chosen according to statistics analysis and domain knowledge. Particularly, we collect 4775 executable malware samples and parse their import tables. All API calls appearing in the import tables are counted, and the most frequently used 260 ones are selected as monitor targets. They are all security relevant. These API calls are used to implement file activities, register activities, process activities, network activities, system service activities, and GUI operations, covering most concerned malicious behavior in malware detection.

To analyze malware behavior in depth, only names of API calls are not enough, arguments are also important. In Windows, calling an API follows the "stdcall" mode; that is, before executing the body of an API, its arguments must be pushed into the function stack from left to right. Then the following call instruction will automatically store the return address at the top of the stack. In x86 architecture, ESP register serves as an indirect memory operand pointing to the top of the stack at any time. Accordingly, return address of an API call can be read from the memory pointed to by ESP. The first argument can then be read from the memory pointed to by ESP+4 and so forth. Especially, for argument whose length exceeds 32 bits, such as strings or structures, ESP adding an offset merely points to another memory space where the real value of this argument is stored. To get this value, it needs to read virtual memory twice or more. But all these are not enough, as defined in MSDN, API arguments can be categorized into three kinds: IN argument, OUT argument and IN_OUT argument. IN argument and the input value of an IN_OUT argument can be read from the calling stack once the entry point of an API is reached, while return value, OUT argument, and the output value of a IN_OUT argument cannot be read until API returns. This means we need a two-stage strategy to get whole API call arguments. Therefore, front-end and back-end callback functions are designed in Osiris. That is, every monitored API call has a corresponding front-end and a corresponding back-end callback function. Once the entry point of an API is reached, its front-end callback function is automatically invoked to read the IN arguments, and after the API returns, its back-end callback function is automatically invoked to read the return value and OUT parameters.

Another problem in API monitoring is how to filter the nested API calls. Nested API calls are inevitable when using the entry point address comparison method to capture API calls, because an API usually invokes another API calls to accomplish compound functionality. For example, CopyFile invokes CreateFile and WriteFile to complete the copy operation. These nested API calls just specify the implementation details of Windows API calls. They do not represent any behavior information of the program under analysis and thus are useless and even harmful to the analysis. To filter these redundant nested API calls, Osiris employs a return address stack in which return address of outermost API call is recorded. Before executing a translation block, Osiris first compares the instruction pointer to the entry addresses of all monitored API calls to decide whether a sensitive API is invoked. If the comparison succeeds, Osiris pushes the address of current translation block into the return address stack, which equals the return address of this API call. If the comparison falls, the instruction pointer is then compared with the address stored at top of the return address stack. If they are equal, it means a monitored API returns. To filter nested API calls, Osiris keeps records of API call information only when the depth of return address stack equals one. Figure 2 is the flowchart to illustrate how API capturing module works.

In summary, the API call capturing module in Osiris is mainly made up of front-end callback functions, back-end callback functions, the framework to invoke callback functions, and the stack of return address. Main functionalities of these components are summarized in Table 1.

4.3. Handling Page Fault. Osiris collects behavior data of analysis process at virtual machine monitor layer silently. A problem worthy of notice when reading API arguments is handling page fault. Windows system uses a lazy policy in memory management [15]; that is, the data may not exist in virtual memory but in virtual hard disk until the first time it is used. Then, it is possible that when Osiris reads IN argument of an API call, once its entry point is reached, the argument may not be in the virtual memory, because it has not been used yet. In this case, if Osiris goes on reading the argument forcibly, an undesirable page fault will be triggered by analyst; as a result, analysis program inside the guest OS will terminate abnormally.

This problem often happens when the argument is a pointer pointing to a large buffer and we need to read contents

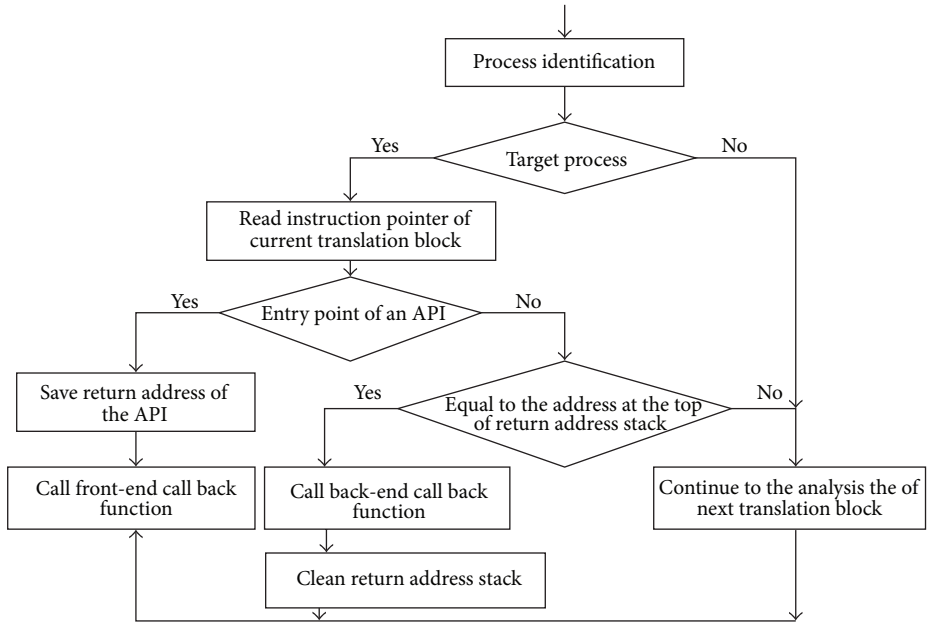


Figure 2: Flowchart of API capturing module.

Table 1: Functionalities of each component in API capturing module.

Module name	Functionality
Framework to invoke callback functions	Comparing the instruction pointer of current translation block to the entry point addresses of monitored API calls; maintaining return address stack; allocating storage buffer for front-end callback functions and back-end callback functions; invoking front-end or back-end callback functions
Front-end callback functions	Being invoked when entry point of a monitored API call is reached; initializing API call arguments capturing environment; reading the address of OUT parameters; reading IN parameters
Back-end callback functions	Being invoked when a monitored API returns; reading OUT parameters, IN_OUT parameters, and return value; writing output information to log files; cleaning buffer

of the buffer, such as the second argument of ReadFile or WriteFile. When using ReadFile or WriteFile to read or write a big file, Windows system may not directly copy the data to the buffer in memory, but only do this the first time the data are used. Osiris gets starting address of the buffer through ESP register; however the buffer may actually be empty at the point when entry point of the API is reached. Reading data which are not in virtual memory triggers a page fault. In order to solve this problem, we use a three-step method.

(1) Before reading the data, test whether it is in the memory. If not, go to step (2).

(2) Wait until the missing page is copied into memory, and then read the data. If it fails, go to step (3).

(3) A program inside guest OS reads data in the missing page by force. This operation will trigger the execution of page fault handling program in guest OS; thus the missing page will be copied into memory. Then Osiris will try to read the data once again.

Nevertheless the above method can effectively avoid page faults that are triggered by analyst; it still requires extra endeavors to do test and wait until missing page is copied into memory. If it can be determined in which situation page fault may occur, it will be a great help to improve analysis efficiency. Actually, stack space in Windows system is nonpaged; they will never be exchanged outside the memory. Besides, string and structure are nonpaged as well. Actually, great majority of the page faults happen in I/O procedure in which big buffer is read or written.

4.4. Monitoring Multiple Processes. After identifying and monitoring behavior of the main process, the next question is how to dynamically and automatically add other analysis targets at the runtime of main process. This is the problem of multiprocess identification.

Based on the above method, it is easy to identify child processes created by main process. Recall that in Windows system, PDB address can be used as identifier of a process and is stored in CR3 register. The value of CR3 register will change as process switches. Before executing the analysis file, Osiris records PDB address and names of all active processes running in the virtualized XP system. While the main process is running, by monitoring CR3 register, Osiris knows when process switches. Once CR3 register is switched to an unknown PDB address, it means a new child process

is created, and then Osiris reads related process information from EPROCESS and monitors its behavior.

Identifying injected processes is a more complex task than child processes, because we must recognize the behavior of process injection at the very time it happens; after that names as well as other necessary information of the injected process can then be obtained. By considering that process, injection is implemented though several API calls and Osiris is mainly designed to capture API calls; naturally by analyzing data dependences between API calls which are related to process injection during runtime of the main process, Osiris recognizes process injection.

The way to inject into another process can further be categorized into two types: injecting into a self-creating process and injecting into already existing system processes. For the former type, the name of injected processes can be extracted though argument of NtCreateProcess which is a kernel mode API to create new process. For the latter type, names of injected process can be extracted by analyzing arguments of API calls which are used to enumerate process. The process identification module maintains a set of global event templates of process injections which track the data dependences between concerned API calls. Take the latter type as an example, when a process enumeration API is invoked, for example, EnumProcess, Process32First, or Process32Next, an event template of process injection is filled for each enumerated process. The template covers process name, PID, process handle, and many other critical information to analyze a process. Although there are various ways to inject into a process, core API calls are relatively stable, including OpenProcess, VirtualAllocEx and WriteProcessMemory. Osiris modifies the front-end or back-end callback functions of these API calls. When they are invoked, all event templates of process injection are indexed and updated. If WriteProcessMemory succeedes, it means that a process injection event occurs. Then we check process name from the relevant event template and take it as index to traverse EPROCESS list to get PDB address of the injected process. Finally passing the obtained PDB address to API capturing module, Osiris successfully adds the injected process as a new target process, which will be monitored afterward.

For service processes, Osiris monitors their behavior in four cases. (1) Main process creates a service and starts to execute it after its creation. (2) Main process creates a service, but it will not be executed until next time computer boots. (3) Main process releases a dll and registers it as a svchost service. The service starts to execute after its creation. (4) Main process releases a dll and registers it as a svchost service. The service will not be executed until next time computer boots. For (1), because the created service is a newly individual process, it makes no difference to monitor behavior of a child process. For (2) and (4), a second time execution is necessary. By analyzing data dependences of API calls, Osiris automatically identifies analysis target, for service that does not be executed; after timeout of main process Osiris starts it as an individual service process or loads it properly as a svchost service to monitor its behavior. For (3), Osiris automatically adds the specific svchost process as a target and then monitors its behavior.

5. Network Simulation

Nowadays, people's life relies more and more on Internet, and so do malwares. They utilize network to infect and spread. It is rare that a malware does not have network activities at all; therefore network activities play an important role in malware analysis. For security consideration, a malware analysis environment should be isolated so that program under analysis would not threaten other computers over the Internet. However, if the analysis environment is totally insulated from Internet, malicious behavior of a malware is sure to fail, leading to insufficient information to judge whether it is malicious. Besides, some malwares are made of a client end and a server end. Usually we can only get the server end, hardly both. Because lacking of control commands, malware behavior performs deficiently. In this situation, we expect to construct bogus responses to network requests to stimulate malicious behaviors of the malware as possible. To achieve these objectives, we develop the multivirtual machine framework which is illustrated in Figure 3.

The analysis environment is mainly made up of three Qemu emulators. First, virtual network cards are installed on the host operating system, each of which is used to communicate with an individual Qemu. Then these three Qemus are bridged together to form a virtual network. The first Qemu is denoted as VM1 in which ROS is installed to redirect network traffic. The second Qemu is denoted as VM2 in which Internet service programs are installed and configured to provide network service and give bogus responses. The last one is a Qemu which is modified according to Section 4 and we denote it as VM3. Actually, more than one modified Qemu (VM3) can be distributed in the virtual network with only one ROS (VM1) and one Qemu to run Internet services (VM2); thus parallel analysis can be performed to improve the efficiency of behavior capturing. The main work to build network environment for malware analysis is done in VM1 and VM2.

To simulate the entire Internet in three Qemu, ROS plays a key role which redirects all network traffic from VM3 to VM2. Particularly, it mainly handles the following five cases (but not restricted to these). First, any DNS request from VM3 is redirected to VM2, and the DNS server in VM2 gives a bogus response with IP address of VM2. This means that any DNS request of the analysis program will get a successful response. Second, if the file under analysis downloads files from a remote FTP server, this file downloading request will be redirected to the FTP server in VM2, and the FTP server provides a preprepared file which is the same type as the requested file. As a result, any file downloading requests will succeed, and so does the file uploading. Third, any Email request from VM3 will be redirected to the email server in VM2. Accordingly, the programs under analysis logs into the email server whether anonymously or with account name and password, the connection will success. Besides, the contents and the attachments are also kept in record to do further analysis. Fourth, any request to visit a web page from VM2 is redirected to the web server in VM3 so that it will always be successful. Fifth, TCP connection to any remote port will be redirected to a predefined port in VM3; this makes

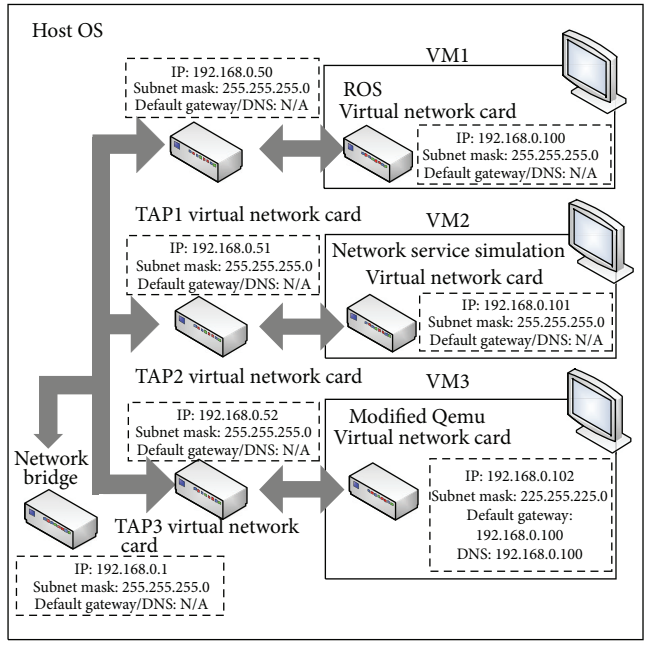


Figure 3: Network environment simulation of Osiris.

Figure 4: Parts of the analysis report for Backdoor.Win32.Hupigon.hypj given by Osiris.

the first TCP connection be successfully established all the time. For the first few TCP packages provide useful information to know about network interaction of a malware, they are important for analysis. Osiris will automatically log the first TPC packages and analyze them. All the redirection and simulation work is transparent for program under analysis.

6. Experiment and Evaluation

To evaluate the ability of Osiris in capturing security relevant behavior of an executable file and also to verify the effectiveness and accuracy of the results, we tested 5169 executable files in Osiris, including 4906 malwares and 263 benign programs. Each test sample runs two minutes. These samples cover virus, Trojan, worms, and backdoors. Osiris analyzes these files in batches and automatically handles possible analysis error. Behavior capturing of all these malware samples finally finished in 7 days and 5 hours. It is demonstrated that it can be used to facilitate large-scale analysis.

Since it is trivial to list all of analysis reports here, we then choose a representative one to explain what Osiris monitors in detail. We choose malware sample Backdoor.Win32.Hupigon.hypj, as it covers almost all of the functional flows of Osiris. Part of the analysis report in XML format can be found in Figure 4. From the report, it can be seen that this malware creates a child process "iexeplore.exe." The child process then creates "calc.exe," and injects into it. Osiris automatically monitors behavior of the child processes and the injected process. Moreover, main process creates a service and starts it next time the computer system boots. Instead, Osiris starts the service after timeout of analyzing the main process. In Figure 4, the process named "continue_Virtualnat.exe" is just the service process.

There exists data dependence between API calls that the return value or output argument of an API can be the input

Figure 5: Detailed behavior of the main process.

Table 2: Some detailed analysis results for backdoor/Win32.Hupigon.hypj.

Behavior	Behavior parameter	
	Description from human analyst	Osiris
Create file	%Program Files%\Common Files\Microsoft Shared\ MSInfo\Virtualnat.exe;	C:\Program Files\Common Files\Microsoft Shared\ MSINFO\Virtualnat.exe
Copy file	×	Source path: C:\Program Files\Common Files\ Microsoft Shared\MSINFO\Virtualnat.exe Target path: C:\Program Files_Virtualnat.exe
Search file	klif.sys	C:\WINDOWS\system32\drivers/klif.sys
Set registry key	HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\ Services\Virtualnat\Description	HKEY_LOCAL_MACHINE\SYSTEM\ CurrentControlSet\Services\Virtualnat
Create process	iexeplore.exe	C:\program files\internet explorer\IEXPLORE.EXE
Create process	calc.exe	C:\WINDOWS\system32\calc.exe
Inject process	iexeplore.exe	iexeplore.exe
Inject process	calc.exe	calc.exe
Create service	×	Image path: C:\Program Files\Common Files\ Microsoft Shared\MSINFO\Virtualnat.exe Description: Virtual Network Control Service
Connect remote port	TCP; port number: 80; IP address: 60.190.92.75	TCP; port number: 80
Open URL	×	http://www.5ai8.net/ip.txt
Create window	×	TApplication

argument of another API. This data dependence is useful to filter API calls which are repeatedly called. It can also be used to consolidate multiple dependent API calls into an independent behavior. Therefore, after obtaining original API calls, Osiris does low-level data dependence analysis to generate a better analysis report. Figure 5 illustrates this idea, which depicts part of the behavior of the main process.

We then compare the report generated by Osiris to behavior description offered by Antiy Labs which is given by manual analysis [16]. Since behavior descriptions given by human expert are from a higher level, they are more concise. By contrast, Osiris is an automatic system to provide behavior data. When executing a program, it is inevitable that there are redundant behavior performed by the operating system. Therefore, analysis report given by Osiris is more detailed and longer. Thus, the comparison is not to prove that Osiris is superior to manual analysis, but to verify the accuracy of results given by Osiris. The comparison results are given in Table 2. In it, "×" means that such behavior is not available.

Because manual analysis only gives behavior descriptions of the main process, thus comparison made in Table 2 only involves main process. By comparison, the conveniences of Osiris system are as follows. First, Osiris successfully monitors all the behavior given by human expert and provides more implementation details. It also gives some behavior not mentioned in manual analysis report. This shows the correctness of the results given by Osiris and also demonstrates that Osiris can be used to provide data to facilitate human decision

Table 3: Comparison of Osiris with other online sandboxes.

	Osiris	Anubis	CFISandbox	Norman Sandbox
Supported file format	PE; pdf, office files; ink; html; eml	PE; URL	exe	exe
Monitoring spawned processes	√	√	√	×
Automatic execution of the released file and created service if they are not executed	√	×	×	×
API calls at different layer	√	√	√	√
Simulated network service	√	×	×	√
Internet access (filtered)	×	√	√	√
Host events simulation	√	×	×	×

making. Second, Osiris automatically records behavior of all processes while the main process is running and even automatically executes the process that will not run until next time the computer boots. It ties its best to provide sufficient information. Besides, many behavior based malware detection algorithms focus on API call sequence; therefore the data collected by Osiris can also be used as original input data by these algorithms [17]. Admittedly, Osiris is only a system to capture behavior data; thus the results given by it are at a lower semantic level. It is not used to replace manual analysis, but to automate the behavior capturing process. Otherwise, a human expert must use a debugger or a disassembly tool to do so.

Furthermore, we compare Osiris with the other online sandboxes, including Anubis, CFISandbox, and Norman sandbox. Table 3 shows the overall comparison results.

Overall speaking, Osiris gives comparable results to these online sandboxes and also has other functionality of its own. It can provide richer behavior data from different sources, including static analysis results, original API call sequences, behavior descriptions, network packages with contents, and visiting logs of server program in Internet service simulation emulator. It can be used in behavior based malware analysis.

7. Conclusions and Future Works

In this paper, we present Osiris to capture malware behavior. It has the following properties. (1) In Osiris, by modifying source code of the open source system emulator Qemu, behavior capturing is implemented at the virtual machine monitor layer which is the most privileged layer in a virtual machine (emulator) system. Therefore, they are more precise for analysis. (2) Osiris captures behavior of all processes which are created at runtime of the main process, including child processes, injected processes and service processes. Furthermore, a twice execution strategy is designed to collect behavior of the process which is executed next time the computer boots. (3) A multivirtual machine framework is proposed to simulate the necessary network environment for malware analysis, so that network behavior of a malware is stimulated to a large extent. (4) Host events are simulated to stimulate the hidden behavior of a malware. Experimental results show that Osiris system can provide rich behavior data for malware analysis. It can be used to facilitate behavior based malware analysis.

Meanwhile, there are still many works to do in future. Currently, Osiris focuses on capturing user mode API calls of a program under analysis. Considering that rootkit infection is also a serious problem, we plan to extend Osiris to monitor behavior of rootkits soon after. Furthermore, Osiris is a system provides original behavior data for behavior based malware analysis; thus the results given by it are at a lower semantic level than manual analysis. Therefore in future, we plan to deeply utilize the data obtained by Osiris. Machine learning or data mining procedure is needed to give further analysis and detection result. Besides, we plan to improve the report of network behavior monitoring. Currently, much information from different sources are obtained but not so well organized, such as log file of the network server programs, as well as network packages. The trivial ones should be filtered out and the information should be formalized. We also plan to put Osiris online soon so that it can be used by other people.

Acknowledgments

The work was jointly supported by the National Natural Science Foundations of China under Grant nos. 61072109, 61272280, 41271447, and 61272195; the Program for New Century Excellent Talents in University (NCET-12-0919); the Fundamental Research Funds for the Central Universities under Grants no. K5051203020 and K5051203001.

References

[1] P. Beaucamps, "Advanced polymorphic techniques," *International Journal of Computer Science*, vol. 2, pp. 194–205, 2007.

[2] E. Filiol, "Malware pattern scanning schemes secure against black-box analysis," *Journal in Computer Virology*, vol. 2, no. 1, pp. 35–50, 2006.

[3] G. Jacob, H. Debar, and E. Filiol, "Behavioral detection of malware: from a survey towards an established taxonomy," *Journal in Computer Virology*, vol. 4, no. 3, pp. 251–266, 2008.

[4] E. Filiol, G. Jacob, and M. Le Liard, "Evaluation methodology and theoretical model for antiviral behavioural detection strategies," *Journal in Computer Virology*, vol. 3, no. 1, pp. 23–37, 2007.

[5] X. Chen, J. Andersen, Z. Morley Mao, M. Bailey, and J. Nazario, "Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware," in *Proceedings of*

International Conference on Dependable Systems and Networks (DSN '08), pp. 177–186, June 2008.

[6] D. Inoue, K. Yoshioka, M. Eto, Y. Hoshizawa, and K. Nakao, "Malware behavior analysis in isolated miniature network for revealing Malware's network activity," in *Proceedings of IEEE International Conference on Communications (ICC '08)*, pp. 1715–1721, May 2008.

[7] G. Hunt and D. Brubacher, "Detours: binary interception of Win32 functions," in *Proceedings of the 3rd USENIX Windows NT Symposium*, pp. 135–143, USENIX Association, 1999.

[8] M. Egele, T. Scholte, E. Kirda et al., "A survey on automated dynamic malware-analysis techniques and tools," *ACM Computing Surveys*, vol. 44, pp. 1–49, 2012.

[9] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of the USENIX Annual Technical Conference*, pp. 41–46, USENIX Associations, 2005.

[10] U. Bayer, C. Kruegel, and E. Kirda, "TTAnalyze: a tool for analyzing malware," in *Proceedings of the 15th Annual Conference on European Institute for Computer Antivirus Research (EICAR '06)*, Hamburg, Germany, 2006.

[11] G. Willems, T. Holz, and F. Freiling, "Toward automated dynamic malware analysis using CWSandbox," *IEEE Security and Privacy*, vol. 5, no. 2, pp. 32–39, 2007.

[12] "Norman sandbox whitepaper," 2012, http://www.norman.com/documents/wpsandbox.pdf.

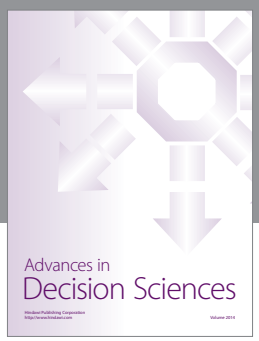
[13] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection and monitoring through VMM-based "out-of-the-box" semantic view reconstruction," *ACM Transactions on Information and System Security*, vol. 13, no. 2, article 12, 2010.

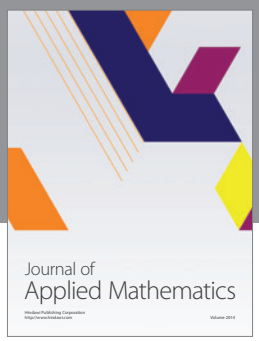
[14] M. Probst, "Dynamic binary translation," in *Proceedings of UKUUG Linux Developers' Conference*, Bristol, UK, 2002.

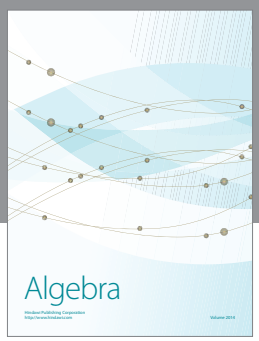
[15] M. E. Russinovich and D. A. Solomon, *Microsoft Windows Internals: Windows Server 2003*, O'Reilly Media, Cambridge, Mass, USA, 2009.

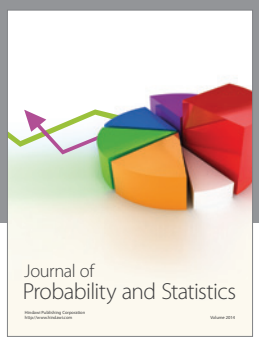
[16] Antiy Labs, 2013, http://www.antiy.com/cn/security/report-more.htm.

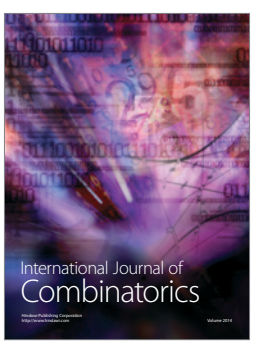
[17] C. Kolbitsch, P. M. Comparetti, C. Kruegel et al., "Effective and efficient malware detection at the end host," in *Proceedings of the 18th Conference on USENIX Security Symposium (USENIX Security '09)*, pp. 351–366, USENIX Association, 2009.

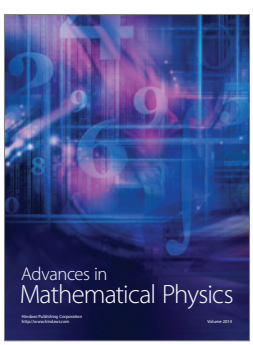


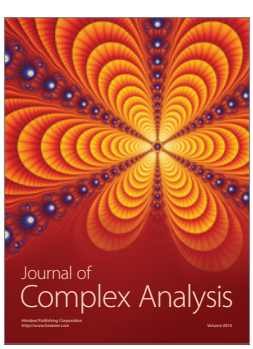


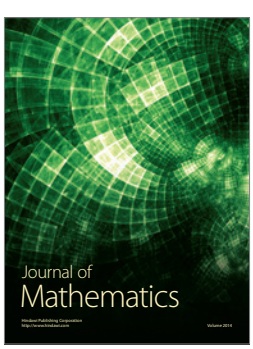


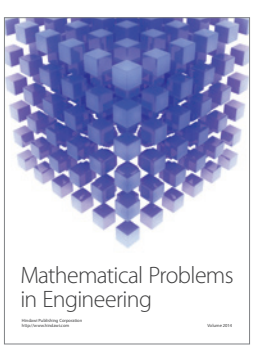


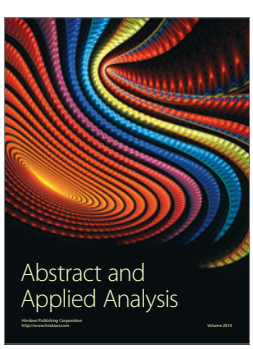


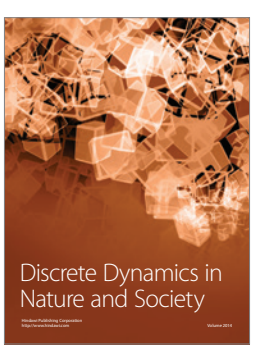


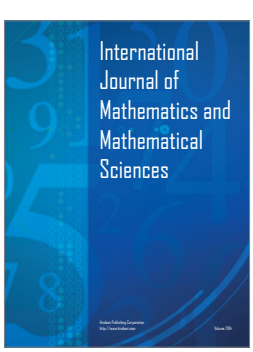












Hindawi

Submit your manuscripts at
http://www.hindawi.com