Identifying System Calls Invoked by Malware using Branch Trace Facilities

Yuto Otsuki, Eiji Takimoto, Shoichi Saito, Eric W. Cooper, and Koichi Mouri

***Abstract*—We are developing Alkanet, a system call tracer for malware analysis. However, recent malware infects other processes. Others consist of two or more modules or plug-ins. It is difficult to trace these malware because traditional methods focus on threads or processes. Getting the system call invoker by stack tracing is a traditional method to solve this problem. However, if malware has falsified its stack, this method cannot identify it correctly. In this paper, we describe a method for identifying a system call invoker by branch trace facilities. We consider the effectiveness of branch trace facilities for malware analysis.**

***Index Terms*—malware analysis, dynamic analysis, branch tracing, virtual machine monitor.**

I. Introduction

MALWARE has become a major security threat on computers. There is already a lot of anti-virus software. However, new kinds or variants of malware emerge by the thousands every day. Anti-virus developers have to enhance their software. To do this, firstly, they need to analyze malware and understand its behavior.

An analysis result should distinguish the behavior of target malware from the behavior of other software. System call tracing, which is to trace system calls invoked by malware, is one malware analysis method. Alkanet [1], a malware analyzer we are developing, uses the method. Conventional system call tracing distinguishes system calls invoked by malware by means of the running process or thread. This method is effective if malware runs as a process or thread. However, recent malware hides its malicious codes in the memory space of the other processes. In this case, a legitimate thread in the process executes the malicious codes. The conventional method cannot distinguish a system call invoked by the malicious codes because the thread which has invoked the system call is legitimate. Other recent malware consists of two or more modules and plug-ins. The conventional method cannot understand which malicious modules invoked the system calls. Therefore, a system call tracer must trace the behavior of the above-mentioned malware without confusing it with the behavior of legitimate software and other malware.

To solve this problem, we have to understand which executable file or generated code has invoked the system call. The main technical issue is how to get the control flow of a thread before the thread invokes a system call. Stack tracing is an existing solution to get a call hierarchy by getting return addresses in the thread's stack. However, if malware has falsified its stack, this method cannot get the correct flow. In addition, stack usage is not uniform. A return address might not point to the caller in some programming techniques or optimizations. Therefore, we propose another solution using the branch trace facilities of recent processors. Our solution generates a call hierarchy based on information of actually-occurring branches. This method is effective to identify the invoker even if malware has falsified the thread's stack.

We have implemented our proposed method in our system call tracer Alkanet. In this paper, we describe a comparison of our method and existing methods using stack tracing. We consider the effectiveness of branch trace facilities for malware analysis.

Manuscript received December 22, 2014; revised January 21, 2015.

Y. Otsuki is with Graduate School of Information Science and Engineering, Ritsumeikan University 1-1-1 Nojihigashi, Kusatsu, Shiga 525-8577 Japan e-mail: yotuki@asl.cs.ritsumei.ac.jp.

S. Saito is with Graduate School of Engineering, Nagoya Institute of Technology.

E. Takimoto, E.W. Cooper and K. Mouri are with College of Information Science and Engineering, Ritsumeikan University.

II. Overview

In this section, firstly, we describe an overview of Alkanet. We also describe an overview of our proposed method to identify an invoker.

A. Alkanet

Alkanet is a system call tracer for malware analysis using VMM. Malware analyzers implemented in VMM can analyze with a higher privilege level than malware. So many anti-debugging techniques would be ineffective for Alkanet. Alkanet can observe running malware without the malware interfering. In addition, malware running in user-mode needs to invoke system calls to affect the environment. Therefore, Alkanet traces invoked system calls and analyzes malware behaviors.

Achievement of system call tracing requires hooking every system call invoked by malware and getting the arguments and return value. In addition, the tracing requires analysis of the meaning of the arguments and return value to get detailed information of system calls invoked.

Malware behavior can be analyzed from the system call logs. Our system extracts a summary of malware behavior from further analysis of the system call logs.

Figure 1 presents the overview of Alkanet. Alkanet is implemented based on BitVisor [2]. BitVisor runs directly on the hardware and does not require a host operating system, and instead runs on processors with Intel Virtualization Technology (a.k.a. Intel VT). Intel VT assists virtualization by VMM. Therefore, BitVisor runs faster than emulators and VMMs implemented in software only. BitVisor can run Windows without requirement of modifications. In addition, BitVisor adopts the parapass-through architecture and does not emulate specific hardware. BitVisor provides the physical hardware for a guest operating system. Therefore, malware cannot detect BitVisor by characteristics of hardware, unlike emulators and VMMs that emulate specific hardware.

ISBN: 978-988-19253-2-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

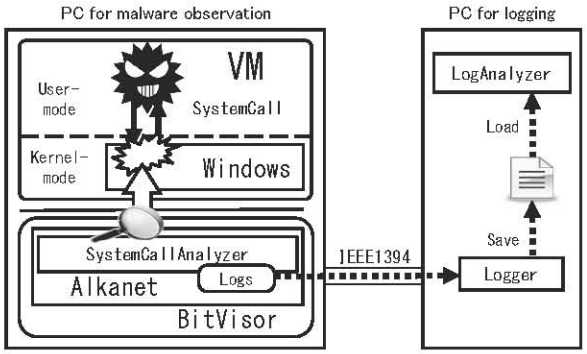

Fig. 1. Overview of Alkanet

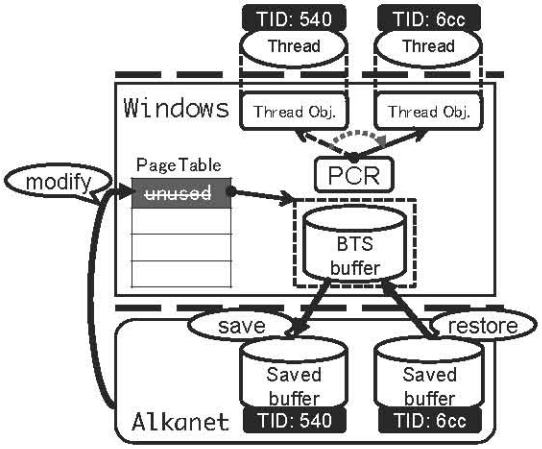

Fig. 2. Alkanet with Branch Trace Store (BTS)

Furthermore, Alkanet adopts Windows XP Service Pack 3 32bit edition as its guest operating system. Alkanet executes malware in this environment. A system call in the environment usually uses the `sysenter` and `sysexit` instructions. `sysenter` enters from user-mode to kernel-mode. `sysexit` returns from kernel-mode to user-mode. To get inputs given to system calls and their results, Alkanet hooks both `sysenter` and `sysexit`. Alkanet uses hardware breakpoints to hook system calls. Alkanet sets breakpoints on entry point and exit point of system call handler. Alkanet hooks invoked system calls and records the number of system calls, their arguments, return values, and so on.

Another machine obtains the system call logs via IEEE 1394 interface. IEEE 1394 has direct read and write access to the physical memory of connected devices. This direct access allows the tracing logs to be obtained without the malware detecting or interfering. Other programs can readily examine the log and process the analysis results to understand intentions of malware behavior.

B. Getting the Invoker

Our method consists of two functions. One function gets the control flow of a thread. The other is to get information of passed files and codes in the flow.

The former function gets information of actually-occurring branches in the VM by the Branch Trace Store (a.k.a. BTS) which is one of branch trace facilities of Intel processor. This function generates the call hierarchy similar to stack tracing by branch records. The latter gets the memory map of a process by Virtual Address Descriptors (a.k.a. VAD) and Page Table Entries. This function finds mapped executable files and generated codes in the memory space of a process. Alkanet can identify the system call coming from malicious executable files or generated codes.

To implement these functions to Alkanet in VMM, there are the following technical issues.

1) Tracing branches occurring in VM.
2) Separating branch records for each thread.
3) Generating a call hierarchy by branch records.
4) Getting the memory map of a process.

III. BTS

Recent Intel processors have branch trace facilities. The Last Branch Record (a.k.a. LBR) saves information of branches in Model Specific Registers (a.k.a. MSR). BTS stores branch records in a memory-resident buffer. Single Step on Branches (a.k.a. BTF) throws a debug exception whenever the processor executes a branch instruction when setting the TF flag in the EFLAGS. Our proposed method uses BTS because its number of stored branches is configurable. The remainder of this section details BTS.

When BTS is enabled, the processor stores a branch record in a buffer in the kernel memory space whenever the processor executes a branch instruction. A branch record contains a source address, a destination address, and a flag indicating whether the branch is predicted.

BTS is configurable for each processor by using Debug Store (DS) mechanism and some MSRs. The DS mechanism provides a DS save area which contains settings for BTS buffer such as virtual address or size. IA32_DS_AREA MSR points to the DS save area. IA32_DEBUGCTL MSR has TR bit (bit 6) and BTS bit (bit 7). The BTS mechanism starts recording branches in the BTS buffer when both these bits are set. IA32_DEBUGCTL MSR has some optional bits. BTS_OFF_OS bit (bit 9) and BTS_OFF_USR bit (bit 10) are for CPL-qualified branch trace mechanisms. It is a filtering function based on the current privilege level (a.k.a. CPL). When BTS_OFF_OS bit is set, processor skips branch recording if CPL is 0. BTS_OFF_USR is for skipping the recording if CPL is greater than 0. When the BTINT bit (bit 8) is set, processor generates an interrupt when the BTS buffer is full. BTINT makes it possible to record branches without limit. On the other hand, when BTINT bit is not set, the BTS buffer is treated as a circular buffer. In this case, of course, processor overwrites from the oldest records.

IV. Getting the Control Flow

This section details a practical use of BTS for VMM to trace processes on VM. It contains solutions for the issue 1 and 2 described in Subsection II-B. Figure 2 shows an overview of our implementation. The remainder of this section gives the details of this figure.

A. Branch tracing on VM

BTS seems useful for getting the control flow of malware. However, our use of BTS creates a gap between a tracer and its targets. The tracer runs as VMM. Alkanet and some other malware analyzers are implemented as VMM to analyze

ISBN: 978-988-19253-2-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

malware stealthily. On the other hand, targets are processes on the VM.

To trace branches occurring from user programs on VM, the VMM must set up a BTS buffer and a DS save area for each processor on the VM. BTS requires virtual addresses for them. In this case, a guest OS must have PTEs for these addresses because VM is running while enabling BTS. As shown in Figure 2, Alkanet modifies Windows's unused PTEs to allocate memory regions for BTS data structures. Windows may use the physical memory allocated for BTS by Alkanet. Alkanet modifies the MmPfnDatabase, which is a data structure to manage physical addresses on Windows. Windows understands the physical memory is unusable.

Alkanet also sets MSRs for BTS on VM. IA32_DEBUGCTL inside VM can be set using Virtual-Machine Control Structures (a.k.a. VMCS). On the other hand, Alkanet sets physical IA32_DS_AREA because VMCS doesn't have an entry for the MSR. Branches in kernel-mode are not necessary because our method's goal is to get the system call invoker. Alkanet sets BTS_OFF_OS bit to record branches occurred in user-mode.

B. Branch tracing for each thread

BTS continues recording branches in the same buffer even when a context switch has occurred. The BTS buffer contains records from some processes or threads if BTS is used simply. To get the control flow of malware, branch records must be separated for each thread. Therefore, Alkanet allocates the BTS buffer for each thread. When a context switch has occurred on Windows, Alkanet switches the buffers accordingly.

To implement this function, Alkanet must hook context switches in Windows. Windows has the Processor Control Region (a.k.a. PCR) for each processor. PCR contains a pointer which points to the object for the current thread. Alkanet detects changes in the pointer by setting a hardware breakpoint to it.

Alkanet saves the buffer for the previous thread and restores the buffer for the next thread, when hooking a thread switch on Windows. Alkanet manages the saved buffer for each thread in the VMM memory space. The saved buffer must be restored when the thread is dispatched again. Windows assigns a unique thread ID (a.k.a. TID) for each thread regardless of its owner process. Basically, a thread is identified by its TID. Therefore, Alkanet determines which saved buffer should be restored, based on TID. Figure 3 shows the case that Windows switches the thread (TID: 0x540) to another (TID: 0x6cc). Alkanet detects the switch by changing the pointer inside PCR. The current BTS buffer in VM is for the thread (TID: 0x540) at this time. Alkanet copies the BTS buffer to a buffer for the thread. It restores the BTS buffer from the saved buffer for the thread (TID: 0x6cc).

The exception is the case which the thread has already been terminated. It is possible that a dead thread and a new thread have same TID because Windows often reuse ownerless TIDs. Alkanet should confirm that the next thread is identical with the thread which is at the time of having saved the buffer, even if their thread ID are same. To do this, Alkanet checks their owner process IDs and the addresses of their object.

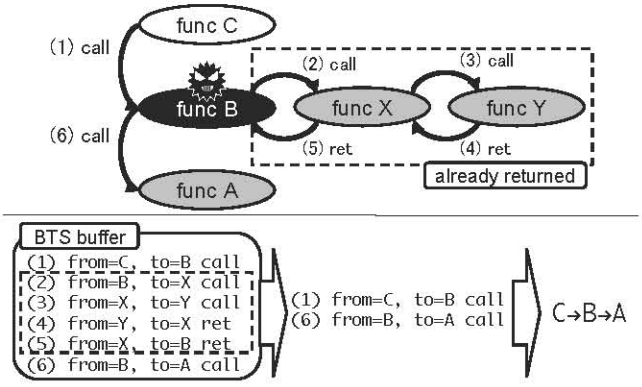

Fig. 3. Generating the call hierarchy by BTS

V. Generating the Call Hierarchy

Section IV gives how to get control flows of user programs on Windows. Alkanet can identify the invoker by generating a call hierarchy based on branch records when hooking a system call. This section details a solution for the issue 3 described in Subsection II-B.

Figure 3 illustrates our method with an example. The example shows a control flow when func A has invoked a system call. The BTS buffer in the example shows branches by `call` or `ret` instructions in the flow. In fact, BTS records all branches including branches from other instructions. In addition, the example shows branch records with instructions but branch instructions are not recorded in the BTS buffer. In this example, func B is in a memory region occupied by malware. The other functions are benign.

Alkanet generates a call hierarchy to identify the invoker when hooking the system call. To do this, Alkanet extracts branches by `call` instructions from all branch records in the BTS buffer. The result is incomplete as a call hierarchy because it contains `call`s which have already returned. In Figure 3, there are already returned `call`s: branch (2) from func B to func X and branch (3) from func X to func Y. These `call`s are to be removed. They have corresponding `ret`s, which have already been executed. Alkanet matches the `ret` branch to each returned `call` in the BTS buffer. Figure 3 contains two pairs of such `call` and `ret`. One of the pairs is branch (3) and branch (4) in a flow between func X and func Y. Another is branch (2) and branch (5) in a flow between func B and func X. A call hierarchy can be generated based on remaining records: branch (1) from func C to func B and branch (6) from func B to func A. The hierarchy shows func A has been reached via B from C. It is equivalent to the result of stack tracing.

Alkanet understands which system calls are invoked by malware based on a generated call hierarchy. In the case of Figure 3, func A is called by func B which is injected by malware. Func B is the root of branches to grayed functions. Even if these grayed functions themselves are not malicious, they organize malicious behavior. Thus, Alkanet identifies the invoker of a system call coming from func A as malware.

VI. Getting the Memory Map

Section IV gives getting a call hierarchy. Func B is included inside a malicious region in the example of Figure 3. In fact, to find the malicious region, Alkanet gets a memory

ISBN: 978-988-19253-2-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

map of a process on Windows. This section gives how to get the memory map and identify malicious codes. This method is a solution for the issue 4 described in Subsection II-B.

Windows manages a memory map of a process using the VAD tree [3]. VAD is a data structure which describes a use of a memory range. A new VAD is created whenever a process allocates new memory region and maps a file. A VAD tree for each process is built by liking other VADs which describes memory space for same process. The process object has a pointer for the root node of the tree. The tree is a self-balancing binary tree based on each VAD's range.

A VAD has the information of a memory range. For example, VAD shows a mapped file in a memory range. It also shows default attributes of pages in the range. Therefore, Alkanet can obtain executable files included in the call hierarchy reaching a system call by retrieving from the VAD tree. If these files includes malicious files, the system call is malicious. In addition, functions can be identified using an export table and symbols of a executable file.

Most malware generates codes dynamically because packed malware unpacks itself dynamically. The other reason is malware often injects codes into other processes. In these cases, malware allocates a memory region by invoking NtAllocateVirtualMemory system call. Windows creates a VAD for the region into the VAD tree of the malware process. Malware modifies the page-protection attributes of the region to writable and executable by NtProtectVirtualMemory system call. Finally, generated codes are written to the region by malware.

The region has certain characteristics. One is that a file is not mapped on the region which is confirmed in the VAD for the region. The region is also writable and dirty. These attributes are contained by the PTEs which point to the region. Malware may remove the attributes after generating codes. In this case, Alkanet can capture this behavior because malware must invoke the NtProtectVirtualMemory system call to modify the attributes. Thus, if a call hierarchy contains addresses matching these characteristics, a system call originates from generated codes.

VII. Evaluation

To confirm the effectiveness of our method, we analyzed a malware sample implemented as DLL using our prototype. This section gives the result of this evaluation.

A. Evaluation method

We implemented not only our method but also traditional stack tracing in the original Alkanet. This evaluation confirms matching the result of the traditional method and a call hierarchy generated by our method. Concretely speaking, our prototype tries to match each return address in a stack to each `call` branch based on the depth, demonstrating that our method is effective for identifying a system call invoker.

In this regard, however, our evaluation prototype doesn't use BTINT. That is to say that the BTS buffer is treated as a circular buffer. The older records are lost by overwriting the newer ones. Therefore, we confirm the effectiveness of our method within the range of only records in the BTS buffer.

B. Log entry

Figure 4 shows part of a log entry, which is described in Subsection VII-C, only the StackTrace in a log entry of system call tracing. The first line shows the information of a thread stack at the time of having hooked a system call. The following, from the second line, describes return addresses and stack frames. A number in "`[]`" is the depth of a stack frame.

The following information is shown for each return address.

a) API: This item shows the name of an API which contains the return address. The name is obtained from the export table or symbols of a mapped file. The item also shows the offset of the head of the API. "-" is shown in the cases without file mappings or symbols for the return address.

b) Writable: This item shows whether the page containing the return address is writable or not.

c) Dirty: This item shows whether the page containing the return address is dirty or not.

d) VAD: This item shows the information of a VAD containing the return address. Specifically, the first inside item shows the range of the VAD. ImageMap is a flag which shows whether the file is mapped. File shows a path of the mapped file if ImageMap is 1.

In addition, "From" and "Valid" provide the result of matching stack tracing and our method. "From" presents a source address of the call branch corresponding with the return address. "Valid" presents YES, NO or UNKNOWN from the matching result. YES is shown if the return address is equal to the "From" address and the length of the `call` instruction. NO is shown in cases in which these values are not equal. If branch records have been lost by the above-mentioned reason, our method cannot generate enough call hierarchy. UNKNOWN is shown in this case.

For example, the return address is 0x7c94d1fc in the `[00]` entry of Figure 4. It was obtained from 0x7ed24 in the stack. "Writable: 0" shows that a page containing the return address is not writable. The page is contained in a VAD which manages a range of from 0x7c940000 to 0x7c9dc000. "ImageMap: 1" shows that a file is mapped in the range. The file is "\WINDOWS\system32\ntdll.dll". The return address is located in NtDelayExecution API, which is a stub for NtDelayExecution system call in ntdll.dll. "From" is 0x7c41d1fa. The return address is equal to the "From" address and the instruction length. Thus, "Valid: YES" is shown.

C. Result of a DLL sample

In this evaluation, we made rundll32.exe load a DLL malware sample into its memory space. The sample is one of the actual instances of malware recorded in CCC DATAset 2013, which is included in MWS Datasets 2014 [4]. We confirm that system calls by malware can be extracted in all logs. Here, we call the sample Conficker.dll based on the name assigned by anti-virus software.

Figure 4 presents a part of a log in this evaluation. Entries for stack frames `[11]` and `[10]` show that rundll32.exe loaded Conficker.dll using LoadLibraryW. Entries `[05]` and `[04]` show that the LdrpCallInitRoutine API called Conficker.dll. Entries `[04]`–`[00]` show that Conficker.dll called

ISBN: 978-988-19253-2-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

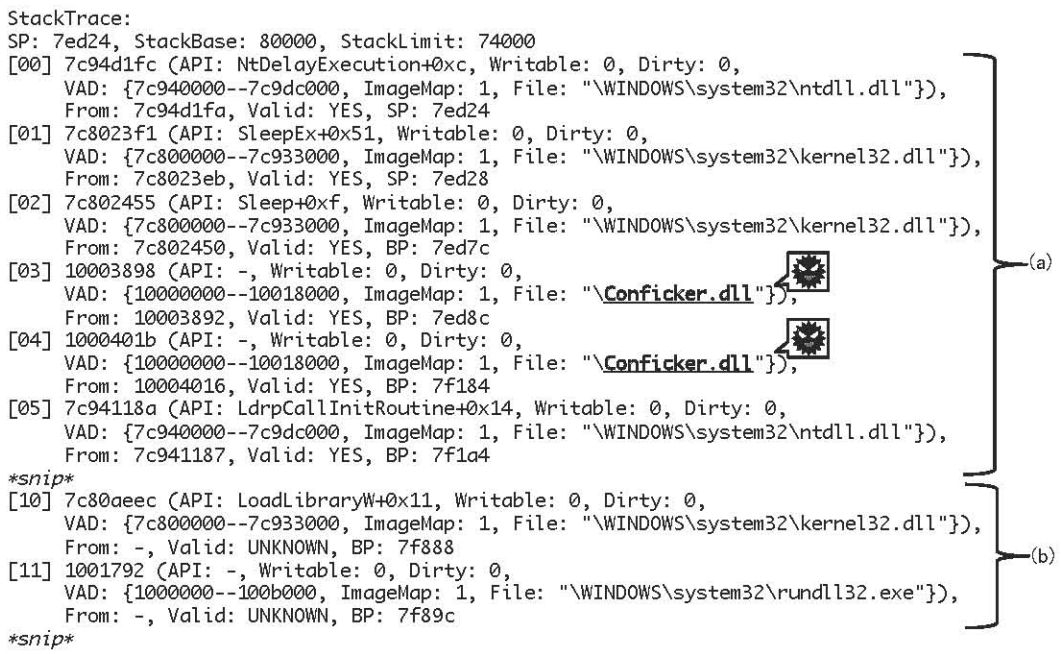

Fig. 4. A log entry of Conficker.dll

the Sleep API. NtDelayExecution system call was reached via SleepEx API and NtDelayExecution stub from Sleep. Thus, the system call originated from Conficker.dll.

Entries (a) in Figure 4 are "Valid: YES". The entry means that each return address and From address in these entries were matched correctly. On the other hand, "Valid" in entries (b), which were located in the deeper positions in the stack, are UNKNOWN because the BTS buffer was limited in this evaluation. This issue can be solved by expanding the buffer whenever a BTINT notification is triggered. Therefore, our method using BTS can obtain the equivalent result to the stack tracing, and effective for identifying the system call invoker.

VIII. Performance

To evaluate slowing down by BTS, we ran PCMark05 System Test Suite [5] on our implementation. The evaluation PC has Intel Core 2 Quad Q6600 2.4GHz and 4 GB memory. This evaluation compared the performance of the four environments as follows.

e) Native: This environment doesn't contain any VMM or tracing methods. Windows runs directly on the evaluation PC.

f) Alkanet (Normal): This is the original Alkanet, which does only system call tracing.

g) Alkanet (ST): This is Alkanet with stack tracing.

h) Alkanet (BTS): This is our evaluation prototype described in Section VII. It matches stack tracing and our method using BTS whenever hooking system calls.

The overall score of Native, Alkanet (Normal) and Alkanet (ST) were 5557, 4171 and 3266, respectively. A unit of the score is PCMarks. The scores of Alkanet (Normal) and Alkanet (ST) are normalized to 75 and 59 respectively. This normalization is based on the score of Native as 100.

On the other hand, the overall score of Alkanet (BTS) was not calculated because it failed Web Page Rendering test, on which Internet Explorer is executed. We concluded that the reason for failure was timeouts of inter-process communication due to slowing down.

Alkanet (BTS) and Native were compared based on the score for each test. On many tests, the scores of Alkanet (BTS) were 10% of the Native's. The exceptions were only tests related with HDD. The causes of slowing down are the overhead of BTS and the following.

1) Hooking a context switch and then saving and restoring the BTS buffer.
2) Generating a call hierarchy and matching it with stack tracing.

Cause 2) appears to bring large slow down, because identifying a branch instruction on each source address of a recorded branch. Our prototype copies branch records between the BTS buffer and saved buffers to switch them. This overhead is a large part of Cause 1). To remove the overhead of memory copies, PTEs for the BTS buffer should be modified to point to the next buffer again.

Just for reference, we evaluated the performance of vanilla QEMU in the same way. Its scores for many tests were 10% of the Native's. As expected, the exceptions were only HDD tests. Thus, the performance of our prototype is comparable with that of QEMU.

IX. Related works

kBouncer [6] mitigates Return-Oriented Programming (a.k.a ROP) attacks by using LBR, which is one of the branch trace facilities on Intel processor. It detects ROP by checking a control flow recorded on LBR when hooking APIs. Its goal is ROP attack detection or mitigation, but not malware analysis. Our method is proposed as a practical use of BTS for malware analysis. In addition, kBouncer is implemented as a driver on Windows. It has no gaps between the tracer and targets. Our method is implemented in VMM to be stealthy for malware and in addition to recording branches occurred on VM.

ISBN: 978-988-19253-2-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

CXPInspector [7] provides a method for hooking intermodular calls, such as flows between the main executable file and DLLs. CXPInspector is implemented in VMM, and modifies executable attributes of pages to implement the hooks. CXPInspector uses LBR to get information of the hooked branch. Our method uses BTS to identify the system call invoker by generating a call hierarchy.

X. Discussions

A. Identifying a system call invoker

We confirmed that our method using BTS obtains an equivalent result to stack tracing. Our method focuses attention on which `call` and `ret` instructions are used in a typical flow between functions. However, in fact, usages of instructions are not uniform, as with stack. `call` and `ret` cannot be matched correctly in some programming techniques or optimizations. We confirmed that our method couldn't obtain the expected call hierarchy in some samples. Concretely speaking, one of the samples uses ROP and the other sample contains shellcodes.

Our method cannot generate a call hierarchy in the following cases.

1) Callbacks from kernel.
2) Non-local exits.
3) Stack modifications.
4) Use of `call` instruction excluding function calls.
5) Use of `ret` instruction excluding function returns.
6) Function calls without a `call` instruction.
7) Function returns without a `ret` instruction.

The above-mentioned cases are categorized into two main types. One of the two categories is the intentional stack modification. This category includes 1)–3). The other category is the special usage of instructions. This category includes 4)–7).

Case 1) often occurs on Windows in particular. Windows provides several kinds of callbacks: Asynchronous Procedure Calls (a.k.a. APC), User-mode Callbacks and so on. In this case, Windows makes a process execute arbitrary tasks by modifying its stack. To return to the context before the callback, the process uses specific APIs or interrupts. Windows modifies the process's stack again to restore the context. Thus, our method cannot match `call` and `ret` on the occasion when a callback has been made or returned.

Flows during any callbacks should be separated from software's original flows. To solve this issue, we focus attention on flows between the kernel and a process in this case. When a callback has occurred, Windows kernel returns to specific user APIs such as KiUserApcDispatcher, KiUserCallbackDispatcher and so on. Occurrence of callbacks can be detected by finding these returns from the kernel. On the other hand, the way of returning from a callback varies according to the kind of callback. For example, APC returns by invoking the NtContinue system call. User-mode Callback has several return methods: NtCallbackReturn system call, XyCallbackReturn API and executing `int 0x2d` directly. Separating callbacks from original flows requires dection of these flows.

Cases 2)–7) are completed all in the user space, in contrast with Case 1). In addition, there are various implementations for them. It is difficult to detect these cases completely and revise a call hierarchy more accurately.

Thus, a call hierarchy should be generated based not only on `call` and `ret` instructions but also the source and destination addresses. Our original goal is to make it possible to identify system calls invoked by malware. Our method does not always have to generate the complete and fine-grained hierarchy to achieve this goal. We focus attention in branches beyond the border of memory regions. The regions are provided by VADs. In addition, if symbols are provided for mapped executable files, ranges of functions can be used as more fine-grained regions. Many Windows DLLs provide symbols. Branches between malware regions and the other regions are important in particular. Detecting the branches makes it possible to identify system calls invoked by malware.

B. Performance

BTS can record all occurring branches. On the other hand, it has a large impact upon performance, as reported in Section VIII. The amount of recorded data increases explosively in some cases. For example, the target program contains loops. Therefore, it is difficult to analyze malware quickly using BTS.

LBR is better suited for quick analysis than BTS because kBouncer [6], described in Section IX, has only a small percent overhead. LBR can choose whether branches are recorded based on the type of branch instructions. Processors since Haswell architecture have the EN_CALLSTACK option for LBR. This option ensures that LBR holds only records for function calls which have not returned yet. LBR with EN_CALLSTACK as an alternative BTS is expected to improve the speed of our proposed method. In this regard, however, LBR has a limitation; the number of branch records is limited by the number of MSRs for LBR. The depth of a call hierarchy which can be generated is limited to sixteen at a maximum because recent processors have sixteen MSRs for LBR. This alternative method has also the issues described in Subsection X-A. We must consider verifying results of both this method and stack tracing.

C. Effective usage of BTS

BTS can capture all control flows of malware because BTS has no limits on the number of branches which can be recorded by enabling BTINT in particular. BTS is better suited for more fine-grained analysis by this characteristic. Concretely speaking, BTS is useful for detection and classification methods based on similarity of control flow. In addition, BTS helps to understand the internal structure of malware. It is effective for specific malware which has a tolerance against static analysis in particular, such as instances in which the malware is packed complicatedly or has a specific loader to load itself. BTS is useful for understanding malware which has new features.

BTS, and also LBR, makes it possible to record branches in physical machines. Emulators have been used to achieve the same goal traditionally. This traditional method has a weak point in malware analysis which is that it is easily detectable by malware because emulators have a lot of characteristics which physical machines do not have. BTS

ISBN: 978-988-19253-2-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

does not have this weak point because it is in the physical machine.

BTS can also help to implement capturing all executed instructions like instruction tracing. Processor executes instructions between a branch and the next one sequentially. The instructions can be obtained from the range determined by records for these branches. Instruction tracing is implemented by single step execution traditionally. BTS makes it possible to implement the method by less frequent hooks. Thus, BTS contributes to improve methods for fine-grained analysis.

XI. Conclusion

In this paper, we describe a method for identifying system call invokers by using BTS. Our method generates a call hierarchy from branch records provided by BTS without depending on the stack. We confirmed that our method could obtain the result equal to one by stack tracing. Our method is effective to identify system calls invoked by malware.

In this regard, however, some issues remain for generating a call hierarchy because `call` instructions and `ret` instructions are not always matched. In addition, unfortunately, BTS has no aptitude for quick analysis because it has a large impact upon performance. BTS would be better off being used for fine-grained analysis, such as taking advantage of capturing all control flows.

In future work, we will focus attention in flows between malware regions and other regions to improve call hierarchy generation. We also consider using LBR with EN_CALLSTACK option.

References

[1] Y. Otsuki, E. Takimoto, T. Kashiyama, S. Saito, E. Cooper, and K. Mouri, "Tracing malicious injected threads using alkanet malware analyzer," in *IAENG Transactions on Engineering Technologies*, ser. Lecture Notes in Electrical Engineering. Springer Netherlands, 2014, vol. 247, pp. 283–299.

[2] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo, and K. Kato, "BitVisor: a thin hypervisor for enforcing i/o device security," in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. ACM, 2009, pp. 121–130.

[3] B. Dolan-Gavitt, "The VAD tree: A process-eye view of physical memory," *Digital Investigation*, vol. 4, pp. 62–64, 2007.

[4] M. Akiyama, M. Kamizono, T. Matsuki, and M. Hatada, "Datasets for anti-malware research ～mws datasets 2014～," in *IPSJ Technical Report Computer Security (CSEC)*, vol. 2014-CSEC-66, no. 19, jun 2014, pp. 1–7, Japanese.

[5] Futuremark Corporation, "Futuremark - Legacy Benchmarks," http://www.futuremark.com/benchmarks/legacy, 2014, accessed 2014-08-25.

[6] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent rop exploit mitigation using indirect branch tracing," in *Proceedings of the 22nd USENIX Conference on Security*, ser. SEC'13. USENIX Association, 2013, pp. 447–462.

[7] C. Willems, R. Hund, and T. Holz, "Hypervisor-based, hardware-assisted system monitoring," in *Virus Bulletin Conference*, 2013.

ISBN: 978-988-19253-2-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)