# Implementation of Virtual Machine Monitor-Based Stack Trace Mechanism on Windows 10 x64

Yuya Yamashita, Junjun Zheng, Shoichi Saito, Eiji Takimoto, Koichi Mouri

***Abstract*—Along with the advent of 64-bit malware, an analysis of such malware is now required. We are developing Alkanet 10, which is a system call tracer using virtualization technology for 64-bit malware analysis on Windows 10 x64. At present, we are attempting to implement a stack trace on Alkanet 10 in order to trace the code injection behaviors of the malware. However, realizing the stack trace is not easy because unlike x86, the calling convention on x64 does not use a frame pointer. We propose implementing the stack trace by using a VAD tree and .pdata section in a PE file.**

***Index Terms*—runtime analysis, stack trace, Windows 10**

## I. Introduction

In recent years, along with the widespread use of 64-bit operating systems, malware has also advanced to 64 bits [1]. In fact, damages caused by 64-bit malware are occurring and are believed to increase in the future. To deal with malware, analyzing its behaviors is necessary, and hence, an analytical environment for 64-bit malware is required. There are two main methods for analyzing malware. One is runtime analysis, which traces the behaviors by actually running the malware, and the other one is static analysis, which analyzes the behaviors by disassembling the codes of malware. Focusing on runtime analysis, we have developed Alkanet which is a system call tracer that allows the analysis of malware on Windows [2]. Alkanet is based on BitVisor which is a virtual machine monitor (VMM) and it can trace system calls invoked by processes on a per-thread basis on Windows XP. We have extended Alkanet to run on Windows 10 [3]. In this paper, we refer to Alkanet for Windows XP x86 as Alkanet XP and Alkanet for Windows 10 x64 as Alkanet 10.

Some malware injects malicious codes into other processes, which are forced to execute these codes in order to hide the behaviors of the malware. In general, system calls or API tracing cannot trace the behaviors of the processes caused by code injections that are associated with each other. Alkanet solves this problem by using stack tracing [4]. By implementing the stack tracing to Alkanet, Alkanet can acquire a function call hierarchy up to system calls. Furthermore, combining the system call tracing log and the call hierarchy would be able to solve this problem.

Alkanet XP implements stack tracing by tracing the values of frame pointers pushed onto the stack in order. However, we cannot apply the same method to Alkanet 10 because the calling convention of Windows 10 x64 does not use frame pointers. So, a method that does not rely on frame pointers is required. In our survey, no study has yet proposed a method of stack tracing on Windows 10 x64. Here we propose a method to realize stack tracing on Windows 10 x64 using a VAD tree which is a internal data structure of Windows and .pdata section in a PE file. We have tested our proposed method and show that stack tracing is possible on Windows 10 x64.

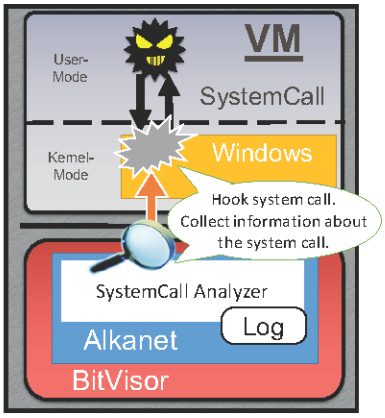


Fig. 1. Overview of Alkanet.

## II. Problems of stack tracing on Alkanet 10

### A. Overview of Alkanet

Figure 1 shows an overview of Alkanet, which is a system call tracer based on BitVisor which is a VMM. Alkanet can trace system calls invoked by malware running on Windows which is a guest OS. Based on a VMM, Alkanet avoids many of the anti-debugging techniques possessed by malware. System call hooking is made by setting hardware breakpoints at the entry and exit of the system call handler. After hooking a system call, Alkanet collects information about system calls such as arguments and return values, then saves the information in a system call tracing log.

### B. How to realize stack tracing on Alkanet XP

Windows XP x86 uses the stdcall calling convention [5] to call the Windows API. All functions must push the value of the current EBP (frame pointer) at the beginning of the function and overwrite EBP with the value of the current ESP. As a result, the area to which EBP is pointing contains the value of the previous EBP value. After 4 bytes, the area contains the return address (see the left-hand diagram in Figure 2). Therefore, stack tracing can be done by sequentially taking out the values of EBP from the stack and reading the address of EBP + 4 each time.

Y. Yamashita, J. Zheng, E. Takimoto, and K. Mouri are with College of Information Science and Engineering, Ritsumeikan University, 1-1-1 Noji-higashi Kusatsu Shiga 525-8577 Japan e-mail: yyamashita@asl.cs.ritsumei.ac.jp.

S. Saito is with Nagoya Institute of Technology, Gokiso Showa Nagoya 466-8555 Japan.

ISBN: 978-988-14048-5-5
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

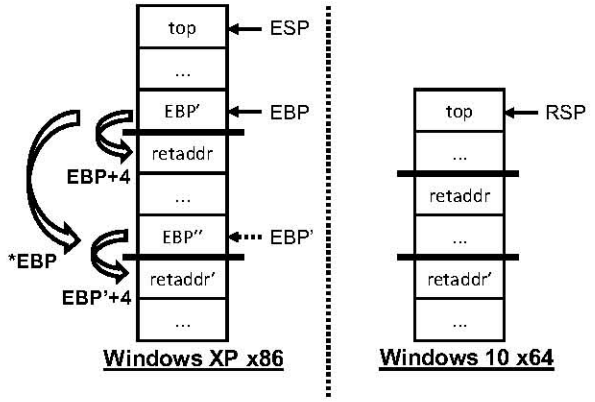


Fig. 2. Stack frame layouts on Windows XP x86 and Windows 10 x64.

### C. Problems of stack tracing on Alkanet 10

Windows 10 x64 uses the Microsoft x64 calling convention [6] to call the Windows API. Unlike the stdcall calling convention, the Microsoft x64 calling convention does not use a frame pointer (see the right-hand diagram in Figure 2). Therefore, a method that does not rely on a frame pointer is required.

## III. Proposed method

### A. Overall design of proposed method

In the Microsoft x64 calling convention, the first few instructions (prolog code) of a function allocate the necessary stack areas, whereas the last few instructions (epilog code) free the stack areas. Therefore, if we obtain the contents of a function's prolog code, we can calculate the stack frame size consumed by the function and implement stack tracing while not relying on a frame pointer. The contents of a function's prolog code are recorded in the .pdata section of the PE file where the function is defined. Our proposed method focuses on this method for stack tracing on Windows 10 x64.

Figure 3 shows the overall design of the proposed method, which consists of the following seven steps.

- Step 1: When hooking, obtain the return address pushed onto the top of the stack.
- Step 2: Search the VAD tree (described later) and find the memory area covering the return address.
- Step 3: From the memory area's information (VAD), obtain the base address and the file name of the PE file where the caller function is defined.
- Step 4: Find the address of the .pdata section of the PE file using the base address.
- Step 5: From the information in the .pdata section, calculate the stack frame size consumed by the caller function.
- Step 6: Using the result of calculating the consumed stack frame size, obtain the next return address on the stack.
- Step 7: Repeat Step 2-6.

We describe the details of the above steps in the following sections.

### B. Details of Step 2

*1) Overview of VAD:* Virtual Address Descriptors (VADs) are internal data structures of Windows 10 for managing the memory areas allocated by processes. A VAD is created every time a process performs a dynamic memory allocation or file mapping and holds the information for managing the corresponding memory area. The information held by a VAD includes the range of the virtual addresses and the file information managed by the VAD.

VADs belonging to the same virtual address space are connected to each other to form a balanced binary search tree (VAD tree). The address of the root node of the tree is stored in the member VadRoot of the EPROCESS structure.

*2) Information required for searching a VAD tree:* The following two pieces of information held by each VAD are required:

- The range of the virtual addresses to be managed
- The address of the child node

Then, we can search the VAD tree as follows.

1) Obtain the return address (RA).
2) Obtain the address of the root node of the VAD tree from EPROCESS.VadRoot.
3) Obtain the virtual address range managed by the VAD of the root node. If this virtual address range includes RA, the search ends. If RA is smaller than the virtual address range, trace the left child node. If RA is larger than the virtual address range, trace the right child node.
4) Trace the child nodes until the target VAD is found.

We can obtain the above two pieces of information from the fields of the MMVAD structure that expresses VAD. Figure 4 shows the fields of the MMVAD and related structures, which we investigated by using the dt command of WinDbg [7]. The meaning of each column of the output in order from left to right is an offset, member name and member type.

*3) How to obtain the range of virtual addresses to be managed:* The range of the virtual addresses managed by VAD is stored in the members of MMVAD.Core. Figure 4 shows the type definition of the MMVAD_SHORT structure which is a type of MMVAD.Core. Among the members of this structure, StartingVpn and StartingVpnHigh hold the start virtual address, whereas EndingVpn and EndingVpnHigh hold the end virtual address. We can obtain the start virtual address by performing the following calculation on StartingVpn and StartingVpnHigh.

$$((StartingVpnHigh << 32)|StartingVpn) << 12$$

We can also obtain the end virtual address by performing the same calculation on EndingVpn and EndingVpnHigh.

*4) How to obtain the address of the child node:* The address of the child node is stored in the members of MMVAD.Core.VadNode. Figure 4 shows the type definition of the RTL_BALANCED_NODE structure which is a type of MMVAD.Core.VadNode. Among the members of this structure, Left holds the address of the left child node and Right holds the address of the right child node, i.e., we can obtain the address of the MMVAD structure of the left child node from Left and of the right child node from Right.

### C. Details of Step 3

*1) How to obtain the base address of the PE file where caller function is defined:* This base address is the same

ISBN: 978-988-14048-5-5
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

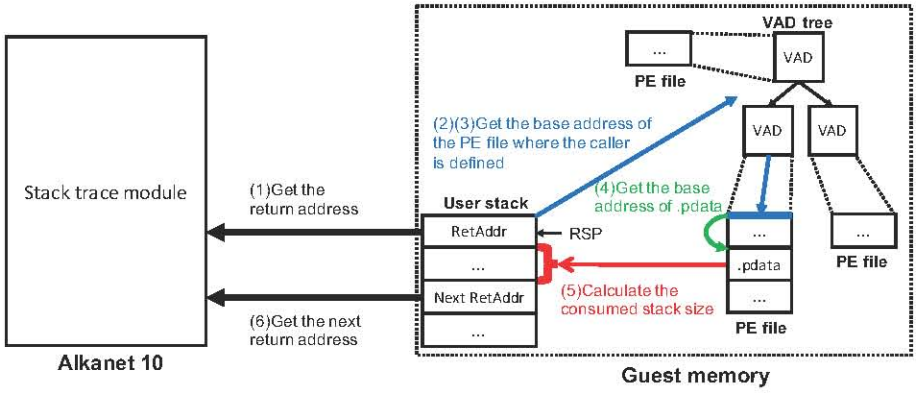


Fig. 3. Overall design of proposed method.

```
kd> dt nt!_MMVAD
   +0x000 Core                 : _MMVAD_SHORT
   +0x040 u2                   : <unnamed-tag>
   +0x048 Subsection           : Ptr64 _SUBSECTION
   +0x050 FirstPrototypePte    : Ptr64 _MMPTE
   +0x058 LastContiguousPte    : Ptr64 _MMPTE
   +0x060 ViewLinks            : _LIST_ENTRY
   +0x070 VadsProcess          : Ptr64 _EPROCESS
   +0x078 u4                   : <unnamed-tag>
   +0x080 FileObject           : Ptr64 _FILE_OBJECT

kd> dt nt!_MMVAD_SHORT
   +0x000 VadNode              : _RTL_BALANCED_NODE
   +0x000 NextVad              : Ptr64 _MMVAD_SHORT
   +0x018 StartingVpn          : Uint4B
   +0x01c EndingVpn            : Uint4B
   +0x020 StartingVpnHigh : UChar
   +0x021 EndingVpnHigh        : UChar
   +0x022 CommitChargeHigh     : UChar
   +0x023 SpareNT64VadUChar    : UChar
   +0x024 ReferenceCount : Int4B
   +0x028 PushLock             : _EX_PUSH_LOCK
   +0x030 u                    : <unnamed-tag>
   +0x034 u1                   : <unnamed-tag>
   +0x038 EventList            : Ptr64 _MI_VAD_EVENT_BLOCK

kd> dt nt!_RTL_BALANCED_NODE
   +0x000 Children             : [2] Ptr64 _RTL_BALANCED_NODE
   +0x000 Left                 : Ptr64 _RTL_BALANCED_NODE
   +0x008 Right                : Ptr64 _RTL_BALANCED_NODE
   +0x010 Red                  : Pos 0, 1 Bit
   +0x010 Balance              : Pos 0, 2 Bits
   +0x010 ParentValue          : Uint8B
```

Fig. 4. Fields of MMVAD, MMVAD_SHORT and RTL_BALANCED_NODE structures.

as the start virtual address managed by the VAD acquired in Step 2. When a process is created from a PE file, the file is mapped onto the memory. At this time, Windows 10 creates one VAD that manages the mapping area, i.e., the start virtual address managed by the VAD created here is the base address of the PE file. The VAD obtained in Step 2 manages the mapping area of the PE file where the caller function is defined because the VAD covers the RA.

*2) How to obtain the filename of the PE file where the caller function is defined:* By following a field of the MMVAD structure obtained in Step 2, we can obtain the file name as follows.

1) Obtain EX_FAST_REF structure from MMVAD.Subsection.ControlArea.FilePointer. Hereinafter, this structure is referred to as efref.
2) Calculate efref.Object AND 0xf, then obtain the address of the corresponding FILE_OBJECT structure. Hereinafter this address is referred to as fo.
3) fo.FileName is the target file name.

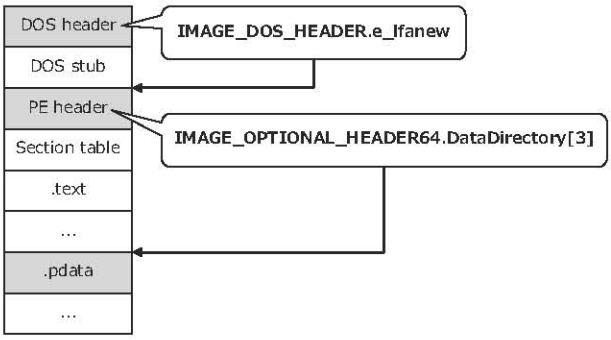


Fig. 5. Overview of Step 4.

### D. Details of Step 4

*1) Overview:* In Step 4, we use the information in the DOS and PE headers included in a PE file. Figure 5 shows the PE32+ format [8] which is a PE file format supporting 64 bits and an overview of Step 4, of which the outline is as follows.

1) Read the field e_lfanew in the DOS header to obtain the start address of the PE header.
2) Read the element of Index 3 of the array DataDirectory which is a field in the PE header and obtain the start address and size of the .pdata section.

In the following sections, we describe the DOS and PE headers related to the above process.

*2) DOS header:* This is an area placed at the head of the PE file and is referred to when executing the PE file on MS-DOS. When executing a PE file on Windows, the fields except for e_lfanew are not used.

Figure 6 shows the structure of the DOS header, which is expressed as an IMAGE_DOS_HEADER structure. The field important to realizing Step 4 is e_lfanew, which holds the Relative Virtual Address (RVA) of the address of the PE header. Here, RVA is an offset from the base address of the PE file, i.e., we can obtain the virtual address of the PE header by adding e lfanew to the base address obtained in Step 2.

*3) PE header:* This is a header that holds a variety of information about a PE file. Figure 6 shows the structure of the PE header, which is expressed as an IMAGE_NT_HEADERS64 structure.

ISBN: 978-988-14048-5-5
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

```
// DOS header
struct IMAGE_DOS_HEADER {
   WORD e_magic;
   WORD e_cblp;
   (snip)
   LONG e_lfanew; // RVA of PE header
};

// PE header
struct IMAGE_NT_HEADERS64 {
   DWORD Signature;
   IMAGE_FILE_HEADER FileHeader;
   IMAGE_OPTIONAL_HEADER OptionalHeader;
};
```

Fig. 6. Structures of DOS header and PE header.

```
struct IMAGE_OPTIONAL_HEADER {
   WORD Magic;
   BYTE MajorLinkerVersion;
   (snip)
   // DataDirectory[3] holds information about .pdata
   IMAGE_DATA_DIRECTORY DataDirectory[16];
};

struct IMAGE_DATA_DIRECTORY {
   DWORD VirtualAddress; // RVA
   DWORD Size;
};
```

Fig. 7. Fields of IMAGE_OPTIONAL_HEADER64 structure and IMAGE_DATA_DIRECTORY structure.

A field important to realizing Step 4 is OptionalHeader. Figure 7 shows the fields of the IMAGE_OPTIONAL_HEADER64 structure, which is a type of OptionalHeader. The RVA and size of the .pdata section are stored in the element of Index 3 in the array DataDirectory. The type of each element in the DataDirectory is an IMAGE_DATA_DIRECTORY structure, which has a VirtualAddress field representing RVA and a Size field representing the size. In addition, the information about the .pdata section is stored in the element of Index 3 in the DataDirectory, i.e., the RVA of the .pdata section is stored in dd3.VirtualAddress and the size of the .pdata section is stored in dd3.Size, where the element of Index 3 in DataDirectory is dd3. Therefore, by reading these two members, we can obtain the start virtual address and size of the .pdata section.

### E. Details of Step 5

*1) Overview:* Using the contents of the prolog code of a caller function, we can calculate the stack frame size consumed by the caller function. We can obtain the contents of the function's prolog code by using the information in the .pdata section. In Step 5, we calculate the stack frame size consumed by a caller function using the information of the .pdata section. Figure 8 shows an overview of Step 5, which consists of the following three processes.

1) Search .pdata section for RUNTIME_FUNCTION structure covering the RA obtained in Step 1.
2) Obtain the address of the corresponding UNWIND_INFO structure from the member of the RUNTIME_FUNCTION structure.
3) Parse the UNWIND_INFO structure and calculate the consumed stack frame size.

*2) Structures of the .pdata section and RUNTIME_FUNCTION structure:* .pdata section consists of an array of RUNTIME_FUNCTION structures [9] whose number of elements is the number of functions defined in the PE file. The RUNTIME_FUNCTION structures have a one-to-one correspondence with the function defined in the PE file and hold information for exception handling related to the corresponding function. This information includes the consumed stack frame size.

Figure 9 shows the member of the RUNTIME_FUNCTION structure. The BeginAddress and EndAddress hold the RVAs of the start and end addresses, respectively, of the function corresponding to the structure. UnwindData holds the RVA of the UNWIND_INFO structure corresponding to the structure. We describe the UNWIND_INFO structure in the next section.

The RUNTIME_FUNCTION structure is placed in the .pdata section in ascending order with respect to the address of the corresponding function, i.e., $EA(n) < BA(n+1)$ always holds, BeginAddress and EndAddress of the nth RUNTIME_FUNCTION structure are $BA(n)$ and $EA(n)$, respectively. Therefore, we can use a binary search to search for the RUNTIME_FUNCTION structure covering the RA.

*3) UNWIND_INFO structure:* The UNWIND_INFO structure [10] is a structure that holds information for the exception handling of the corresponding function. Figure 9 shows the fields of the UNWIND_INFO structure.

The field necessary for realizing Step 5 is UnwindCodesArray, which is an array of the UNWIND_CODE structures. Each element represents one instruction in the prolog code of the function, i.e., we can obtain the instructions of the function's prolog code from UnwindCodesArray, and calculate the consumed stack frame size of the function.

*4) UNWIND_CODE structure:* The UNWIND_CODE structure [11] is a structure for representing one instruction in the prolog code. Figure 9 shows the fields of the UNWIND_CODE structure.

We can calculate the consumed stack frame size by using the fields, UnwindOperationCode and OperationInfo, which store the type of the corresponding instruction. Table 1 shows the relationship between the machine instructions, UnwindOperationCode and OperationInfo. The next slot in Table 1 represents the value of the next element in the UnwindCodesArray. Similarly, the next two slots represents the values of the next two elements in the UnwindCodesArray. In this way, we can obtain the instructions in the prolog code from the UNWIND_CODE structure and calculate the reduction amount of the RSP in the instructions. Therefore, we can calculate the sum of decreases in RSP in the prolog code, i.e., the consumed stack frame size of the function, by iterating UnwindCodesArray and parsing the UNWIND_CODE structure.

### F. Details of Step 6

Figure 10 shows the relationship between the stack frame layout and consumed stack frame size. We can obtain the next RA on the stack by performing the following processes.

1) Add 8 to the RSP.

ISBN: 978-988-14048-5-5
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

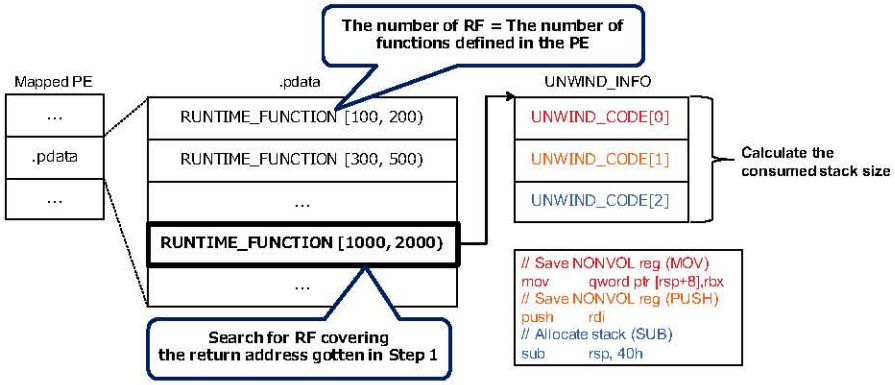


Fig. 8. Overview of Step 5.

TABLE I
RELATIONSHIP BETWEEN MACHINE INSTRUCTIONS, UNWINDOPERATIONCODE AND OPERATIONINFO (ONLY PRIMARY ONES).

| UnwindOperationCode | Reduced amount of RSP | Meaning |
|---|---|---|
| UWOP_PUSH_NONVOL (0) | 8 bytes | Push a nonvolatile register (PUSH) |
| UWOP_ALLOC_LARGE (1) | OpInfo=0: next slot * 8 bytes (range: 136 to 512K - 8)<br>OpInfo=1: next two slots * 8 bytes (range: 512K to 4G - 8) | Allocate an area on the stack (SUB) |
| UWOP_ALLOC_SMALL (2) | OpInfo * 8 + 8 bytes (range: 8 to 128) | Allocate an area on the stack (SUB) |
| UWOP_SAVE_NONVOL (4) | 0 bytes | Save a nonvolatile register on the stack (MOV) |

```
struct RUNTIME_FUNCTION {
   ULONG BeginAddress;
   ULONG EndAddress;
   ULONG UnwindData;
};

struct UNWIND_INFO {
   UBYTE Version : 3;
   UBYTE Flags : 5;
   UBYTE SizeOfProlog;
   UBYTE CountOfUnwindCodes;
   UBYTE FrameRegister : 4;
   UBYTE FrameRegisterOffsetScaled : 4;
   USHORT UnwindCodesArray[n];
};

struct UNWIND_CODE {
   UBYTE OffsetInProlog;
   UBYTE UnwindOperationCode : 4;
   UBYTE OperationInfo : 4;
};
```

Fig. 9. Fields of RUNTIME_FUNCTION, UNWIND_INFO, and UNWIND_CODE structures.

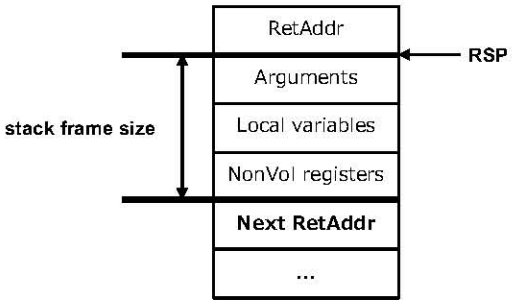


Fig. 10. Relationship between stack frame layout and consumed stack frame size.

2) Add the consumed stack frame size obtained in Step 5 to the RSP.
3) Read 8 bytes from the address to which the RSP points. The result of this reading is the next RA.

## IV. EVALUATION

### A. Purpose and methods

The purpose of this evaluation was to confirm that the stack trace on Windows 10 x64 could be realized by using the proposed method. We confirmed the trace by focusing on the fact that WinDbg had a stack trace function.

In this evaluation, we implemented the proposed method in Alkanet 10 so as to call the stack trace processes when hooking the SYSCALL instruction, after which Alkanet 10 outputs the RVA of the RA and the name of the mapped file of each function call hierarchy. By using these two pieces of information, we performed the evaluations.

Below, we describe the specific method for the evaluations. The method is divided into two parts.

**First part: Stack trace with WinDbg**

1) Make a test program that creates a file using CreateFile API, and attaches WinDbg to the process.
2) Set a breakpoint at ntdll!NtCreateFile, which is the system call stub of the NtCreateFile system call. CreateFile API eventually calls the NtCreateFile system call. For this confirmation, we focus on the function call hierarchy until the NtCreateFile system call.
3) Execute the test program. As a result, WinDbg breaks at the beginning of ntdll!NtWriteFile.
4) Perform the stack trace function of WinDbg and save the result.

**Second part: Stack trace with the proposed method**

1) Activate Alkanet 10 to trace the test program.
2) Save the result of the stack trace.
3) Convert each retrieved RA to the corresponding function name in the result. We can realize this conversion by using the RVA of the RA, the name of the mapped file and the symbol information included in the PDB file.
4) Confirm if the obtained result matches the result of Step (4) in the first part of the evaluation.

ISBN: 978-988-14048-5-5
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

```
// After attaching WinDbg to the test program (file.exe)
0:000> bp ntdll!NtCreateFile
0:000> g
Breakpoint 0 hit
ntdll!NtCreateFile
0:000> knf
// The result is snipped partially
 #      Call Site
00      ntdll!NtCreateFile
01      KERNELBASE!CreateFileInternal+0x2f6
02      KERNELBASE!CreateFileA+0xa8
03      file!main+0x4a
04      file!invoke_main+0x22
05      file!__scrt_common_main_seh+0x11d (Inline Function)
06      KERNEL32!BaseThreadInitThunk+0x14
07      ntdll!RtlUserThreadStart+0x21
```

Fig. 11. Result of first part of evaluation.

```
// ntdll!NtCreateFile (00)
[ALKANET] #3 file_name: \Windows\System32\ntdll.dll
[ALKANET] #3 retaddr_rva: a0924

// KERNELBASE!CreateFileInternal+0x2f6 (01)
[ALKANET] #3 file_name: \Windows\System32\KernelBase.dll
[ALKANET] #3 retaddr_rva: 41bb6

// KERNELBASE!CreateFileA+0xa8 (02)
[ALKANET] #3 file_name: \Windows\System32\KernelBase.dll
[ALKANET] #3 retaddr_rva: 41808

// file!main+0x4a (03)
[ALKANET] #3 file_name: \Users\yyamashita\Desktop\file.exe
[ALKANET] #3 retaddr_rva: 623a

// file!invoke_main+0x22 (04)
[ALKANET] #3 file_name: \Users\yyamashita\Desktop\file.exe
[ALKANET] #3 retaddr_rva: 68a5

// Alkanet 10 cannot trace file!__scrt_common_main_seh+0x11d (05)
// because it is an inline function.

// KERNEL32!BaseThreadInitThunk+0x14 (06)
[ALKANET] #3 file_name: \Windows\System32\kernel32.dll
[ALKANET] #3 retaddr_rva: 11fe4

// ntdll!RtlUserThreadStart+0x21 (07)
[ALKANET] #3 file_name: \Windows\System32\ntdll.dll
[ALKANET] #3 retaddr_rva: 6efc1
```

Fig. 12. Result of second part of evaluation.

### B. Results and discussion

Figure 11 shows the result of the first part of the evaluation. The function call hierarchy until the NtCreateFile system call consisted of eight functions from ntdll!RtlUserThreadStart to ntdll!NtCreateFile. The inline functions and lambda expressions were also traced.

Figure 12 shows the result of the second part of the evaluation. The stack trace with the proposed method was able to trace all functions except for the inline functions correctly. An inline function cannot be traced because the CALL instruction is not used for calling an inline function and the RA is not pushed onto the stack. However, [12] states that the information for tracing inline functions is stored in a PDB file. Therefore, the tracing of inline functions can be realized by utilizing the information in a PDB file if necessary. From the above results, we confirmed that the proposed method was effective for stack tracing on Windows 10 x64.

## V. Conclusion

We proposed a method to realize a stack trace on Windows 10 x64 using virtual machine monitor. By using the information from a VAD tree and the .pdata section, this method does not rely on a frame pointer. We evaluated the method by implementing it on Alkanet 10 and confirmed that it could perform a stack trace correctly.

In the future, we will continue to evaluate the performance of the proposed method. Specifically, we will measure the overhead by comparing the performance of the original Alkanet 10 to that of Alkanet 10 with our proposed method.

## References

[1] Deep Instinct, "Beware of the 64-bit malware," http://info.deepinstinct.com/whitepaper-beware-of-the-64-bit-malware, 2017.

[2] Y. Otsuki, E. Takimoto, T. Kashiyama, S. Saito, E. W. Cooper, and K. Mouri, "Alkanet: a dynamic malware analyzer based on virtual machine monitor," in *2012 World Congress on Engineering and Computer Science, WCECS 2012*. Newswood Limited, 2012, pp. 36–44.

[3] Y. Otsuki, S. Nakano, S. Aketa, E. Takimoto, S. Saito, and K. Mouri, "Implementation of system call tracer for windows 10 x64," *Computer security symposium 2015*, vol. 2015, no. 3, pp. 839–846, 2015.

[4] Y. Otsuki, E. Takimoto, S. Saito, and K. Mouri, "A method for identifying system call invoker in dynamic link library," *Computer security symposium 2013*, vol. 2013, no. 4, pp. 753–760, 2013.

[5] Microsoft, "__stdcall," https://msdn.microsoft.com/en-us/library/zxk0tw93.aspx, 2017.

[6] ——, "x64 software conventions," https://msdn.microsoft.com/en-US/library/7kcdt6fy.aspx, 2017.

[7] ——, "Download windows debugger (windbg) tools - windows hardware dev center," https://developer.microsoft.com/en-us/windows/hardware/download-windbg, 2017.

[8] ——, "Microsoft portable executable and common object file format specification," https://www.microsoft.com/en-us/download/details.aspx?id=19509, 2017.

[9] ——, "struct runtime_function," https://msdn.microsoft.com/en-us/library/ft9x1kdx.aspx, 2017.

[10] ——, "struct unwind_info," https://msdn.microsoft.com/en-us/library/ft9x1kdx.asp://msdn.microsoft.com/en-us/library/ddssxxy8.aspx, 2017.

[11] ——, "struct unwind_code," https://msdn.microsoft.com/en-us/library/ck9asaa9.aspx, 2017.

[12] ——, "Debugging optimized code and inline functions — microsoft docs," https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/debugging-optimized-code-and-inline-functions-external, 2017.