AIT: A method for operating system kernel function call graph generation with a virtualization technique

Longlong Jiao[1], Senlin Luo[1], Wangtong Liu[1] and Limin Pan[1*]
[1] Information System & Security and Countermeasures Experiments Center, Beijing Institute of Technology
Beijing 100081, China
[e-mail: xiguazzz@foxmail.com, luosenlin2012@gmail.com, lwt1231234@126.com, panlimin2016@gmail.com]
*Corresponding author: Limin Pan

Received May 27, 2019; revised December 11, 2019; accepted February 19, 2020;
published May 31, 2020

Abstract

Operating system (OS) kernel function call graphs have been widely used in OS analysis and defense. However, most existing methods and tools for generating function call graphs are designed for application programs, and cannot be used for generating OS kernel function call graphs. This paper proposes a virtualization-based call graph generation method called Acquire in Trap (AIT). When target kernel functions are called, AIT dynamically initiates a system trap with the help of a virtualization technique. It then analyzes and records the calling relationships for trap handling by traversing the kernel stacks and the code space. Our experimental results show that the proposed method is feasible for both Linux and Windows OSs, including 32 and 64-bit versions, with high recall and precision rates. AIT is independent of the source code, compiler and OS kernel architecture, and is a universal method for generating OS kernel function call graphs.

Keywords: Function call graph, operating system kernel, virtualization, system trap

1. Introduction

Operating system (OS) kernel function call graphs have been widely used in OS analysis and defense. Malware-like rootkits that use kernel-level attacks have been increasingly popular in recent years. Malware often gains the highest privilege of the operating system by exploiting vulnerabilities in the OS kernel [1], and can carry out malicious behavior such as hiding, destruction, monitoring, etc. [2-4], and several methods have been proposed to defend against this. Function call graphs play an important role in these methods, such as finding malware hooking code [5], extending the system to defend against ROP attacks [6] and analyzing the similarities between two types of malware [7].

At present, methods for generating function call graphs can be divided into two classes: static and dynamic generation methods. Static methods mainly include the analysis of source [8] and binary code [9]. These types of call graph generation method can achieve a high recall rate, since the instructions for each function call must appear in the entire source file or binary file. However, static generation methods face a significant problem in that dynamic calling, such as via function pointers, is based upon approximations [10], and the analysis is therefore not dependent upon the exact calling relation of the running program.

Dynamic generation methods acquire the function calling relation by dynamically analyzing issues while the target program is running. These methods include inserting testing code into the head of the call function to hook the calling operation [11], obtaining calling information via the Intel Pin framework [12], etc. With the help of the information obtained from dynamic analysis, these generation methods can acquire the exact calling relation of the running program. This characteristic allows for a high precision rate, especially for dynamic calling such as via function pointers. However, the recall rate will be influenced by the comprehensiveness of the testing steps. In view of the pervasive use of dynamic calling in the OS kernel, which cannot be handled accurately by static generation methods, this paper focuses on dynamic generation methods.

The more recent and familiar methods of dynamic function call graph generation focus on the generation of graphs for application programs such as JavaScript programs [10,13], and rely on the use of source code [11], symbol tables [14], a specified compiler, kernel architecture [15-16], etc. However, when applied to the OS kernel, these conditions may not be met. For instance, source code and symbol tables are unobtainable for closed-source OS kernels.

Existing methods of dynamic generation of function call graphs involve many dependency conditions when building the OS kernel function call graph. This paper proposes a dynamic method for generating function call graphs called Acquire in Trap (AIT), which uses virtualization. This method can efficiently build a function call graph for an OS kernel independent of the source code, compiler or OS kernel architecture, and can be used for both Linux and Windows OS, including 32 and 64-bit versions.

The remainder of this paper is organized as follows. Section 2 examines the related work in the field of function call graph generation. Section 3 describes the principles and implementation of AIT with virtualization technology. Section 4 presents an experimental evaluation, and Section 5 discusses the results. Finally, conclusions and future work are drawn in Section 6.

2. Related Work

This paper focuses on dynamic generation methods. The basic architecture for dynamic generation methods is illustrated in **Fig. 1**.Current dynamic generation methods include the classes mentioned below.

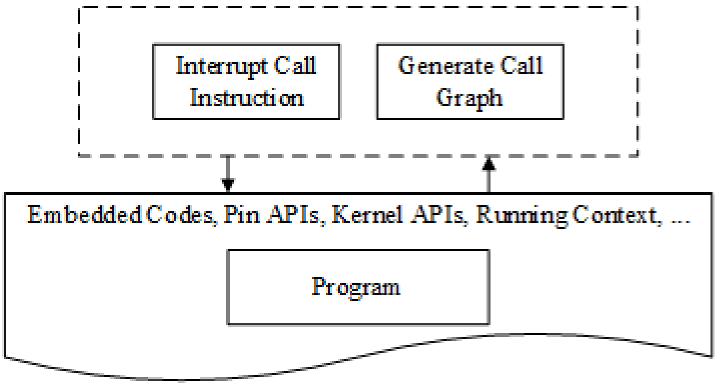
Interrupt Call Instruction
Generate Call Graph
Embedded Codes, Pin APIs, Kernel APIs, Running Context, ...
Program

Fig. 1. Basic architecture for dynamic generation methods

Generation methods implemented using compilers have restrictions on the compiling environment. Some compilers provide options for programmers to add custom codes while compiling, allowing them to change the execution flow by adding codes into the head of each function and recording the calling operation in these additional codes. In iComnstance, GCC provides the "-pg" compile option for programmers to realize this purpose. Based on this, the application analysis tool Gprof adds a function named "mcount" into the head of each function in the program. Whenever a function calling operation takes place, the function "mcount" will execute first, and Gprof will record the caller and callee information. This type of generation method relies on the compiling environment (compiler and source code), meaning that it is unavailable for an OS that is not compiled by these specific compilers, such as Windows. Similarly, other tools that use compiler options to realize OS kernel analysis, such as Ftrace [17], are also limited by the compiling environment.

Generation methods implemented using a kernel API are limited by the kernel architecture. The debug tool Systemtap uses Linux kernel API Kprobes to monitor and track the execution of a program [15]. Kprobes realizes function analysis by hooking target codes at the OS kernel level. Based on Kprobes, Systemtap can analyze the application function calling relation; however, since Kprobes is provided by the Linux kernel, Systemtap can only generate the function call graph for applications in Linux.

Generation methods such as those that use the Pin mechanism can only be utilized in user-level program analysis. Jalan built a framework called Trin-Trin to analyze function calling relations using the Pin mechanism [18]. Pin is provided by Intel CPU for the purpose of analyzing programs, and allows programmers to conFig. interrupt conditions by executing specific piece of code to analyze the running state. Trin-Trin uses this mechanism to interrupt each calling operation, and thus acquires the calling relation. However, as Pin is located in the OS [19], this method can only be used to analyze user-level programs and is thus unavailable for OS kernel analysis.

Generation methods implemented using a stack and register analysis are not accurate for OS kernel investigations. Some debug tools such as Windbg offer checking of the calling route, and provide a calling route analysis for current debugging functions. Windbg implements this function by checking kernel stacks and the EBP register. In most application programs, the return address and EBP value of the caller function will be stored on the stack for the callee function. Windbg uses this information to get the calling relation of the current debugging function. However, many OS kernel functions do not store the EBP value on the kernel stack, meaning that Windbg cannot obtain the accurate calling route. Analysis tools such as Perf [20] analyze the context to obtain information on the caller function for the current function using periodic sampling, but in the same way as Windbg, this process will cause errors when analyzing some operating system kernel functions. Thus, Perf needs the symbol table as a reference, or needs to add extra parameters in compiling, making it unsuitable for certain closed source operating systems.

Existing function call graph generation methods therefore mainly focus on application programs rather than relying on other conditions such as the compiler and kernel architecture. These limitations make these methods inapplicable for OS kernel function analysis. In addition, some dynamic analysis tools and methods rely on the OS kernel API or architecture, meaning that they are only suitable for specific OS kernels or programs.

Following the development of virtualization technology, cloud services based on virtualization platforms have become widely used [21]. At the same time, virtualization technology has generated new opportunities and challenges for the generation of function call graphs. The virtual machine monitor layer has been introduced into the computer architecture of the virtualization platform, and provides a trusted virtual environment that can directly detect and control the running state, kernel code and data of operating system kernel. Using this property, many detection methods based on virtualization platforms have been proposed and studied [22-23].

This paper proposes a universal dynamic generation method for OS kernel function call graphs, called AIT. This approach is based on virtualization technology and does not rely on factors such as the source code, compiler, OS kernel API or architecture. AIT is therefore applicable to many types of OS kernel, including Windows or Linux with 32 and 64 bits.

3. Call Graph Generation Method

3.1 Framework

AIT uses virtualization technology to generate call graphs. By replacing the first two bytes of the target function with specific instructions, the kernel control flow traps into the virtualization space and analyzes the calling relation whenever the target function is called.

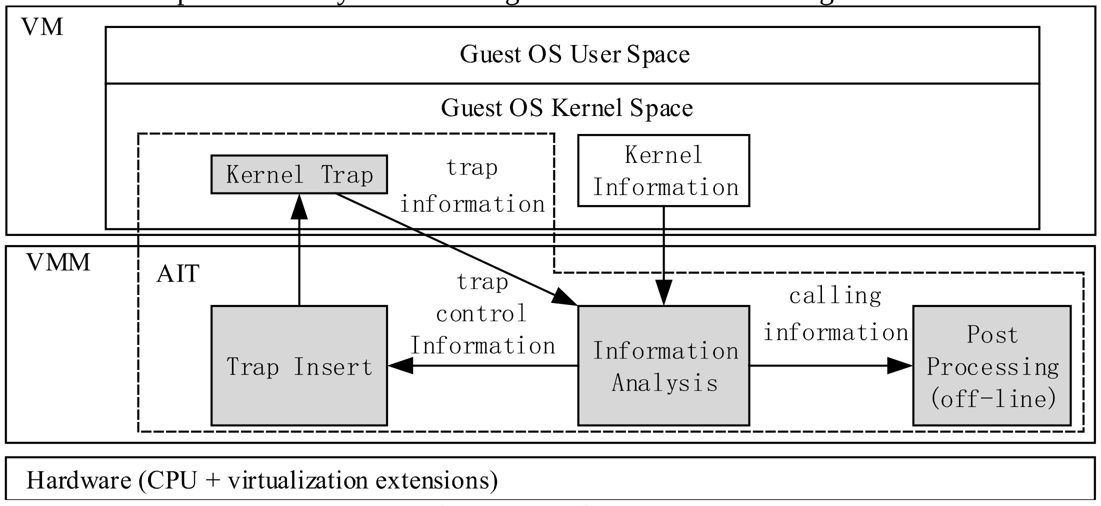

Fig. 2. AIT architecture

The overall architecture of AIT is illustrated in **Fig. 2**, showing the three levels of hardware, virtual machine monitor and virtual machine (with the guest OS kernel).

The guest OS runs on an abstract computing platform called a virtual machine (VM). The middle level, which manages the VM, is called the virtual machine monitor (VMM); this isolates different VMs and has a higher privilege than the guest OS in VM [24], and can therefore analyze and monitor the guest OS kernel.

There are three primary modules in AIT: the trap insert, information analysis and post-processing modules. With the help of the higher privilege provided by the VMM, the trap insert and information analysis modules can monitor the running guest OS kernel. The methods of analysis include modifying kernel instructions and acquiring kernel data and register values. By modifying the kernel instructions, the trap insert module can insert special instructions into the kernel space that form the kernel trap in the guest OS kernel, and trigger the trap handler at a specific time. In the trap handler, the information analysis module determines the function calling information based on the trap and kernel information. Using this information, the information analysis module provides detailed trap control information for the trap insert module. The data and control flow between the trap insert module, the kernel trap instructions and the information analysis module allows for the complete analysis of function calling information. The information analysis module also provides calling information for the post-processing module, which stores the information with a specific structure and handles the duplication issue.

AIT realizes the main functions in VMM rather than the guest OS, and thus the implementation of AIT does not have a close relation with the guest OS as in existing generation methods. In this paper, the open-source project XEN [25] is used to realize AIT. The details of each primary module are described in Sections 3.1 to 3.4.

3.2. Trap Insert

3.2.1. Principles

Using virtualization technology, a trap mechanism is proposed to address various special events such as page faults and interrupts. More specifically, special "trap instructions" such as 0xcc (int3) trigger the trap handler. In the trap handler, the VM and guest OS are paused, and the VMM obtains the control flow and permission to read and write the data or instructions into the kernel space of the guest OS. Using this mechanism, AIT inserts "trap instructions" into important areas of the guest OS kernel, and when the kernel control flow reaches these areas, AIT can pause the guest OS and analyze it.

In this implementation, in order to ensure the normal operation of VM, AIT inserts the trap instruction 0xcc to trigger the trap handler and restore the primeval kernel instruction after kernel analysis in the trap handler. Thus, for a target instruction address T_A to be analyzed, AIT saves the primeval instruction C_A and replaces it with 0xcc. When the kernel instruction flow reaches T_A, 0xcc will be executed and causes the int3 trap. In the int3 trap handler, the information analysis module will complete the analysis, and C_A will be replaced with T_A, after which the guest OS runs normally.

Since the int3 trap mechanism is created by the virtualization framework and the only modification to the guest OS is the replacement of the original code with 0xcc, the trap insert module does not have a significant relationship with the guest OS, providing a high level of compatibility with different OSs.

3.2.2. Implementation

In the open-source hardware virtualization project XEN, the functions hvm_copy_from_guest_virt and hvm_copy_to_guest_virt are provided for reading and writing data in the kernel space of the guest OS. Based on the information presented above, trap insert module in AIT inserts and manages the trap instructions.

In this module, each trap instruction contains two bytes. The first byte is 0xcc, which causes the int3 trap, as described above. The second byte marks this type of trap, as shown in **Table 1**. When a trap occurs, the information analysis and trap insert modules will perform different types of analysis based on the value of the second byte.

Table 1. Type of trap instruction

Value	Description
0x00	Trap in the head of the function
0x01	Trap next to the head of the function
0x02	Trap in the calling instruction
0x03	Trap next to the calling instruction
0x04	Trap in the return instruction
0x05	Trap in the debug address
0x06	Trap in the dynamic calling instruction
0x07	Trap next to the dynamic calling instruction

For each inserted trap instruction, the trap insert module stores the first two bytes and the address of the target code before replacing them with a suitable trap instruction. Following this, when the instruction flow reaches the target address, the 0xcc will cause the int3 trap, and the trap handler in VMM will be executed to deal with the calling information analysis.

AIT needs a function header address as a "starting point". This address is entered as a parameter when AIT is started, and can be obtained from the operating system kernel symbol table or kernel debugging. The header addresses for the rest of the functions can be found automatically from the information analysis module.

After analyzing the calling information, in order to ensure the guest OS can run properly, the replaced instruction must be restored to the stored primeval code. However, in the case of multiple calling relations, the trap instruction should be maintained in order to continue the process of acquiring calling information. A method called "cross-replacing" is used for this purpose in AIT.

We assume that in function A, the first instruction is C1, and the second is C2. The primeval instructions C1 and C2 are first stored as a back-up and C1 is then replaced with 0xcc00, which means this trap is in the head of the function (this is usually the first instruction). After analyzing the information, 0xcc00 is replaced with C1 and C2 is set to 0xcc01, meaning that this trap is next to the head of the function (usually the second instruction). Thus, when the guest OS is resumed, code C1 will also be executed, and a trap is placed at 0xcc01. Finally, 0xcc01 is restored back to C2 and 0xcc00 to C1; the guest OS can then run while the trap remains in the head of the function. Type value 0x01, 0x03 and 0x07 in **Table 1** is in charge of "cross-replacing".

It should be noted that each trap instruction holds two bytes in the guest OS kernel space; if the instruction C1 is shorter than 2 bytes, then 0xcc00 will cover two instructions. Thus, when 0xcc01 is restored back to C2 and replace C1 as 0xcc00, the first byte of C2 will still be replaced as 00 by 0xcc00. As a result, cross-replacing must be executed between the first instruction C1 and the third instruction C3. In order to confirm the length of each instruction in the kernel space, a simple disassembling function is added to the trap insert module in order to test the length of the assembly instruction.

3.3. Information Analysis

3.3.1. Principles

In virtualization technology, the VMM is responsible for providing an abstract hardware platform, meaning that the VMM controls the hardware utilized by VM, including the registers and RAM. Hence, the VMM has the highest permission to read and write data in the guest OS kernel space. Based on this authority, AIT analyzes the information in the kernel space during trap handling.

The information analysis module mainly deals with two targets, the calling instruction address and the function header address. As mentioned in Section 3.2.1, the trap insert module is responsible for inserting the trap instructions into the target address to create the int3 trap. For a target function F_A, the function header is denoted by H_A; if a trap takes place in H_A, F_A is called via a calling instruction C_B, and the address of C_{B+1} (the next instruction after C_B) is stored on the top of the stack. Thus, the information analysis module will acquire the pair $\left(C_B, H_A\right)$ by analyzing the register and the kernel stack.

If the calling instruction C_B belongs to the function F_B, then F_B also needs to be analyzed to complete the entire call graph. In normal circumstances, a function header can be easily analyzed with the help of the OS kernel symbol table. To ensure compatibility with certain OSs that do not provide a kernel symbol table, AIT provides a special method for locating the function header. The information analysis module finds the return instruction R_B from C_B and instructs the trap insert module to set a trap instruction there. When a trap occurs in R_B, which means function F_B will return to its caller function F_C, the return address (the address of C_{C+1}) is stored on the top of the stack. The calling instruction C_C can be found from C_{C+1}, and C_C carries the information of the F_B header address H_B. By following these steps, the relation pair is acquired as $\left(H_B, C_B\right)$.

The complete information analysis relation is shown in **Fig. 3**. The arrows show the call/return direction of the instruction flow, the solid line shows the process of analysis of the pair $\left(C_B, H_A\right)$, and the dotted line shows the process of analysis of the pair $\left(H_B, C_B\right)$. The complete calling relation between F_A and F_B is acquired by combining these two relation pairs as $\left(F_A\left(H_A\right), F_B\left(H_B, C_B\right)\right)$; this means that function F_B (starting from header H_B) uses instruction C_B to call function F_B (starting from header H_A).

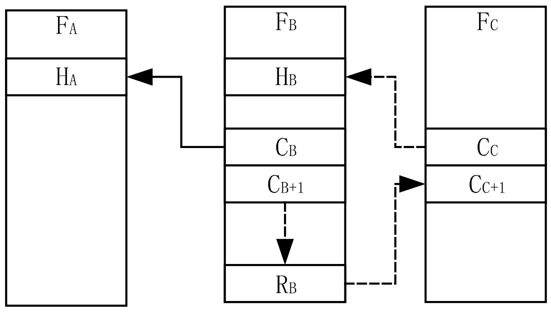

Fig. 3. Information analysis relations

3.3.2. Implementation

In the open-source hardware virtualization project XEN, the int3 trap is processed in the function vmx_vmexit_handler, which is rewritten to realize the information analysis module.

As described in Section 3.3.1, the purpose of the information analysis module is to acquire the calling relation pairs of the target function. This module consists of two main parts: calling relation analysis and trap instruction control.

The calling relation analysis is responsible for acquiring calling relation pairs, such as (C_B, H_A) and (H_B, C_B), as described in Section 3.2.2. The VMM has the highest level of privilege, so the kernel data can be read using the function hvm_copy_from_guest_virt, and the register can be read using function __vmread and regs structure.

In the information analysis module, the functions that need to be analyzed are stored in a function array containing the function header address. The steps involved in a typical analysis are as follows.

(1) The information analysis module instructs the trap insert module to insert the trap instruction into the target functions $(F_1, F_2, \ldots, F_n)$ using the "trap in head of function" type;

(2) When the trap takes place in the head of the function F_A, the caller instruction C_B can be found by analyzing the ESP register and the kernel stack data;

(3) In order to find the header address H_B of F_B, the information analysis module finds the return instruction (0xc3 or 0xc2) from C_B and instructs the trap insert module to insert a trap instruction of the "Trap in return instruction" type;

(4) When a trap occurs in the return instruction, the caller instruction C_C can be found by analyzing the ESP register and the kernel stack data;

(5) By analyzing C_C, the header address H_B of F_B can be found and added into the function table. Finally, the analysis module moves to the next target function.

By following these steps, a multi-level call graph can be generated from a single root function. **Fig. 3** in Section 3.2.2 shows the relations between the symbols, and **Fig. 4** shows how the call graph can be generated. When the analysis of a new target function is complete, the new function is added into the call graph (shown as a dashed box in each step), where the arrow shows the calling direction. It should be noted that the termination condition of the above automated generation process is that no new function call relationship is found within the timeout period.

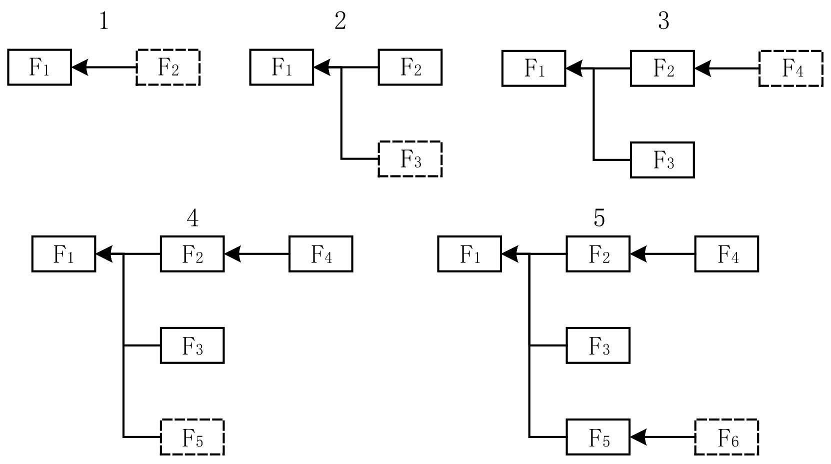

Fig. 4. Call graph generation process

There are two problems that require special attention in the abovementioned steps. Firstly, for some functions with a branch structure, there may be several return instructions (0xc3 or 0xc2) in the function; however, only one will be executed when the function returns. Thus, in Step 3, all of the return instructions in the function need to be found to ensure that the trap will occur when the function returns. The information analysis module therefore searches all the return instructions after C_B by traversing the instruction flow, including the return instructions in the branch structure. It is worth noting that some special functions will not follow the direction of the increasing address, and may jump to a lower address. Thus, the search for the return instruction should not only follow the address direction, but also each jump operation.

In practice, since kernel instructions are stored as assembly instructions, not all functions will have an obvious dividing mark such as 0xcccccc, making it hard to judge the end of the function. In this case, the return instruction is found via the following steps:

(1) Find the first return instruction R_1 from C_B, and insert the trap instruction in R_1 and C_B;

(2) Keep the guest OS running, and wait for a trap to occur;

(3) If the next trap occurs in $R_n\left(n=1,2,3,\ldots\right)$, meaning that the return operation is captured, find the calling instruction C_C, as mentioned above;

(4) If the next trap occurs in C_B, meaning that the return operation has been missed, find the next return instruction R_{m+1} from R_m onwards (R_m is the final return instruction which has been found); then insert the trap instruction in R_{m+1} and return to Step 2.

Second, in Step 5, the calling instruction C_C that carries the information about H_B may be of different types (**Table 2** shows some examples of types of calling instruction).

In some calling instructions, the calling target may be stored statically or may change while the guest OS is running. This means that the target address can only be determined as the calling instruction is being executed. In order to obtain the real calling target address from the calling instruction, a trap instruction is inserted into this calling instruction (C_C) with the "trap in dynamic calling instruction" symbol (0x06 in **Table 1**). When a trap of this type occurs, the information analysis module will record the value of the real calling target address.

Meanwhile, for certain dynamic calling instructions such as call eax, the target calling address may change when the instruction is executed; however, only one target address is the real header address H_B. Thus, all feasible target addresses are collected when the "trap in dynamic calling instruction" trap occurs. In consideration of the real header address, F_B must be in front of C_B, and the most probable target address is selected, which is located before but closest to C_B as the final header address of F_B.

Table 2. Examples of calling instructions

Calling instruction	**Meaning**
0xff15+4bytes address	Call [4bytes address]
0xff5348	Call [ebx+0x48]
0xff55ec	Call [ebp-0x14]
0xffd7	Call edi
0xffd0	Call eax
0xff10	Call [eax]
0xffd3	Call ebx
0xffd1	Call ecx

3.4. Post-Processing

3.4.1. Principles

The function calling information provided by the information analysis module is stored as $\left(F_A\left(H_A\right), F_B\left(H_B, C_B\right)\right)$. Since H_A can represent F_A, and H_B can represent F_B, the information structure can be rewritten in the form of triples $\left(H_A, C_B, H_B\right)$ containing the complete calling information of one function calling operation, i.e. the header address H_A of the callee function, the calling instruction address C_B and the caller header address H_B, as shown in the image marked Simple 1 in **Fig. 5**.

However, for complete analysis processing, the information analysis module will provide a number of calling information triples, such as $\left(H_{A1}, C_{B1}, H_{B1}\right)$, $\left(H_{A2}, C_{B2}, H_{B2}\right)$, $\ldots$, $\left(H_{An}, C_{Bn}, H_{Bn}\right)$. The post-processing module is in charge of cleaning up this information in the following ways:

(1) Dropping duplicate triples. If two triples are identical, they describe the same calling operation at different times, which is very common. Thus, when the information analysis module creates a calling information triple, the post-processing module first tests whether or not this triple is new.

(2) Combining the same callee function. For two triples $\left(H_{A1}, C_{B1}, H_{B1}\right)$ and $\left(H_{A2}, C_{B2}, H_{B2}\right)$, if $H_{A1} = H_{A2} = H_A$, the same function F_A is called by different two calling instructions C_{B1} and C_{B2}. Thus, these two triples should be combined as $\left(H_{A1}, \left(C_{B1}, H_{B1}\right), \left(C_{B2}, H_{B2}\right)\right)$, as shown in the Simple 2 scheme in **Fig. 5**.

(3) Combining the same caller functions. For two triples $\left(H_{A1}, C_{B1}, H_{B1}\right)$ and $\left(H_{A2}, C_{B2}, H_{B2}\right)$, if $H_{A1} = H_{A2} = H_A$ and $H_{B1} = H_{B2} = H_B$, the same function F_A is called by

the same function F_B via different calling instructions C_{B1} and C_{B2}, a situation that will occur in functions with a branch. Thus, these two triples should be combined as $\left(H_{A1},\left(C_{B1},C_{B2}\right),H_{B2}\right)$, as shown in the Simple 3 scheme in **Fig. 5**.

(4) Combining dynamic calling. For two triples $\left(H_{A1},C_{B1},H_{B1}\right)$ and $\left(H_{A2},C_{B2},H_{B2}\right)$, if $H_{A1} \neq H_{A2}$ and $C_{B1}=C_{B2}=C_B$, this means that the calling instruction C_B is a dynamic call (like a function pointer), which will call different functions each time according to its parameter values. Thus, these two triples should be combined as $\left(\left(H_{A1},H_{A2}\right),C_{B1},H_{B1}\right)$, as shown in Simple 4 in **Fig. 5**.

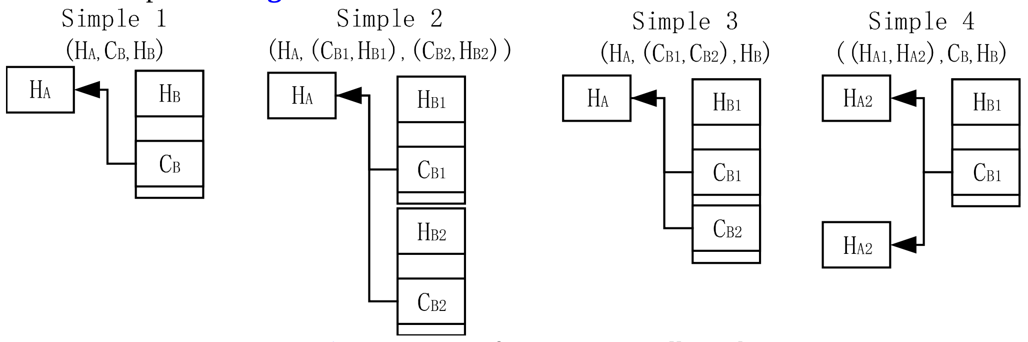

Fig. 5. Process of generating a call graph

3.4.2. Implementation

As mentioned above, the post-processing module is mainly in charge of combining calling information based on the calling information triple provided by the information analysis module.

The calling information triple $\left(H_A,C_B,H_B\right)$ is stored in a two-dimensional chain table. One dimension provides a structure for the information of the callee function, and the other for the the caller function. The information in each dimension is shown in **Tables 3** and **4**.

Table 3. Content of the callee function information

Item	Content
Header address	The header address of this callee funtion
Next header address	Address of second instruction
Header instruction	First instruction
Second instruction	Second instruction
Next callee function	Pointer to next callee function information

Table 4. Content of the caller function information

Item	Content
Calling instruction address	The address of calling instruction
Next calling instruction address	Address of instruction next to calling instruction
Header address	Header address of this caller function

When the information analysis module provides a new calling information triple (H_A, C_B, H_B) the post-processing module completes the processing using the following steps:

(1) Traverse the callee function information chain table, and find the node with the header address H_A;

(2) Traverse the caller function information chain table of H_A, and determine whether C_B is in the table; if not, add a new node with calling instruction address C_B;

(3) Traverse the callee function information chain table, and determine whether H_B is in the table; if not, add a new node with header address H_B to start finding the parent functions of H_B.

Fig. 6 shows how the link list is built while the information analysis module is running.

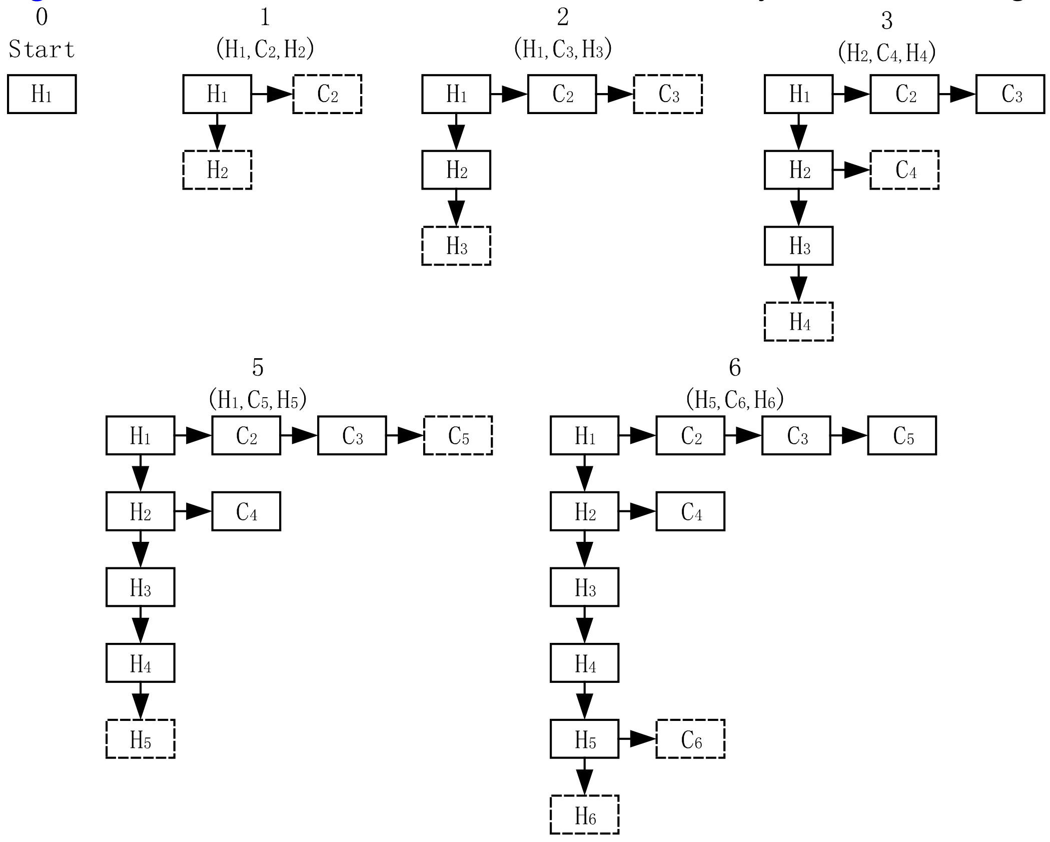

Fig. 6. Process of generating the calling information chain table

4. Experimental Results and Analysis

The purpose of this experiment is to verify the correctness of AIT and to determine whether it is suitable for different OSs. A total of 207 common kernel functions were chosen from 32-bit Windows XP (SP3), 64-bit Windows10 (17134.345) and 64-bit CentOS 7.5, with Linux 3.10.0-862 as the "starting point" and a timeout of five minutes.

A trusted call graph is needed, meaning that the source code of the testing OS kernel is requried. Thus, the Windows Research Kernel (WRK) [26] and Linux 3.10.0-862 kernel source were downloaded to build a trusted call graph for Windows XP and CentOS 7.5. The trusted call graph for Windows 10 was built by manual kernel debugging.

The precision rate and recall rate were chosen as evaluation criteria. The precision rate (*P*) and recall rate (*R*) were calculated using the following formulae:

$$P = \frac{F_D \cap F_H}{F_D} \tag{1}$$

$$R = \frac{F_D \cap F_H}{F_H} \tag{2}$$

where F_D is the number of function call behaviors obtained by AIT and F_H is the number of function call behaviors obtained by hand, with the help of the Windows Research Kernel, the Linux kernel source code and the OS kernel debug tools.

The final results are shown in **Table 5** (where ST means the symbol table of the operating system kernel).

Table 5. Precision and recall results

Operating System	F_D	F_H	$F_D \cap F_H$	P	R
XP SP3 with ST	432	452	432	100%	95.58%
XP SP3 without ST	357	452	357	100%	78.98%
Windows10 with ST	335	351	335	100%	95.44%
Windows10 without ST	247	351	247	100%	70.37%
CentOS 7.5 with ST	379	394	379	100%	96.19%
CentOS 7.5 without ST	285	394	285	100%	72.34%

The results show that the AIT method of call graph generation is suitable for Windows/Linux OS kernel functions, for both 32- and 64-bit OSs. Moreover, the main modules of AIT are in the VMM, meaning that the implementation of these modules barely relates to the guest OS. Thus, when generating different OS kernel function call graphs, the only aspects which need to be modified are the parameters of AIT, as the lengths of the variables depend on the version of the OS.

The calculated precision rate for every function is 100%, since the function calling operation was acquired from the actual execution, meaning that every calling operation that was discovered was executed during the testing process.

The recall rate was above 95% when a symbol table was available. After rechecking the missing function calling relations, it was found that AIT did not traverse the entire code space. Certain special calling instructions will only execute in special cases, such as registry editing or file downloading, and AIT missed these function calling relations that were not activated.

The recall rate dropped significantly when a symbol table was unavailable. After manual debugging, several special kernel functions were found that prevent the function information analysis process in AIT from taking place. These functions mean that AIT cannot get information such as the function header address in order to start the detection of the calling relation, as described in Section 3.2.1.

(1) Some functions are designed as an endless loop. For example, ExpWorkerThread is in charge of managing all of the threads, and has an endless loop to carry out this management,

never returning to its parent function. This will terminate the information analysis process in Section 3.2.2.

(2) Some functions do not have a return instruction (0xc2 or 0xc3). For example, HalpApcInterrupt ends in a special way: pushing the real return address A, relevant parameter and jumping to another instruction address B. Thus when B is complete, the flow will return to A.

(3) The return instruction in some functions cannot be executed. For example, nt!PspExitThread makes the kernel switch the thread before the instruction flow come to the return instruction. This function will also then be interrupted, and the return instruction will never be executed.

These special functions are mainly written directly in assembly language, and programmers simplify the instructions for optimizing the OS kernel. These functions do not follow the standard rules, and the process of analysis of these functions also needs to be specially designed.

AIT, the method proposed in this paper for generating OS kernel function call graphs, has the following advantages. Firstly, the call graph generation method relies only on the mechanism of the function calling process, and does not rely on any other factors except the processor architecture (apply to mainstream CPUs such as Intel). In comparison with existing research studies of call graph methods, AIT has a wider range of applications since it relies only on the source code and a compiler.

Secondly, virtualization technology is used in AIT to analyze the OS kernel, meaning that the generation method is not relevant to the target OS and does not rely on the OS kernel architecture or functions. Hence, AIT is compatible with a wider range of OS kernels.

Thirdly, function calling operations are captured by inserting the trap instruction into the target function header; the parent functions can therefore be acquired easily, which is important in generating a calling whitelist and in kernel security.

However, certain functions such as those mentioned in Section 4 need to be resolved in order to complete the function information acquiring process. In addition, for dynamic calling (such as call [eax]) in the information analysis module, the real header address is determined by collecting every feasible target address, as described in Section 3.3. In some cases, the target function may never be called during the process of information collection, meaning that AIT cannot obtain the real header address and may provide an incorrect header.

Table 6. Functional comparison of results

Generation method/tools	WinDbg	GDB	Ftrace	Systemtap	Pin	Perf	AIT
Not dependent on the compiler	√	√	×	√	√	√	√
Not dependent on the OS type	√	√	√	×	√	√	√
Available for OS kernel function	√	√	√	√	×	√	√
Not dependent on kernel stack backtracking	√	√	√	√	√	×	√
Not dependent on manual debugging	×	×	√	√	√	√	√

5. Conclusions

This paper presents an operating system kernel function call graph generation method called Acquire in Trap (AIT), which generates function calling relations for OS kernels by using virtualization technology to insert trap instructions with different symbols into the important address, and finishes the analysis work in the trap handler to analyze the function calling relations.

Compared with other generation methods proposed in various research studies, AIT is independent of the source code, compiler or OS kernel architecture, allowing for a wide range of applications. The experimental results show that AIT can acquire the OS kernel function calling relations and the function header address information to build the OS kernel functions, resulting in a precision rate of 100% and a recall rate of 87.5%, and is compatible with x86/x64 architectures for both Linux and Windows OS.

Future studies will first focus on extending the practical scope of application of AIT, such as to macOS and FreeBSD. The detection of kernel-level malicious behavior will then be studied based on AIT.

References

[1] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel, "Ensuring operating system kernel integrity with OSck," *ACM SIGPLAN Notices*, vol. 46, no. 3, pp. 279-290, 2011. Article (CrossRef Link)

[2] S. Eresheim, R. Luh, and S. Schrittwieser, "The evolution of process hiding techniques in malware-current threats and possible countermeasures," *Journal of Information Processing*, vol. 25 no.1, pp. 866-874, 2017. Article (CrossRef Link)

[3] A. Singh and K. Chatterjee, "Cloud security issues and challenges: A survey," *Journal of Network and Computer Applications*, vol. 79, pp. 88-115, 2017. Article (CrossRef Link)

[4] C. Cui, Y. Wu, Y. Li, et al., "Lightweight intrusion detection of rootkit with VMI-based driver separation mechanism," *KSII Transactions on Internet & Information Systems*, vol. 11, no.3, pp. 1722-1741, 2017. Article (CrossRef Link)

[5] R. Patil and C. Modi, "An exhaustive survey on security concerns and solutions at different components of virtualization," *ACM Computing Surveys*, vol. 52, no. 1, p. 12, 2019. Article (CrossRef Link)

[6] P. Bhat and K. Dutta, "A survey on various threats and current state of security in Android platform," *ACM Computing Surveys*, vol. 52, no. 1, p. 21, 2019. Article (CrossRef Link)

[7] R. Luh, H. Janicke, and S. Schrittwieser, "AIDIS: Detecting and classifying anomalous behavior in ubiquitous kernel processes," *Computers & Security*, vol. 84, pp. 120-147, 2019. Article (CrossRef Link)

[8] A. Arusoaie, S. Ciobâca, V. Craciun, et al., "A comparison of open-source static analysis tools for vulnerability detection in C/C++ code," in *Proc. of 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, Timisoara, Romania*, pp. 161-168, 21-24 September 2017. Article (CrossRef Link)

[9] A. Lee, A. Payne , and T. Atkison, "A review of popular reverse engineering tools from a novice perspective," in *Proc. of the International Conference on Software Engineering Research and Practice, Las Vegas, Nevada, USA*, pp. 68-74, 30 July-2 August 2018. Article (CrossRef Link)

[10] T. R. Toma and M. S. Islam, "An efficient mechanism of generating call graph for Javascript using dynamic analysis in web application," in *Proc. of 2014 International Conference on Informatics, Electronics & Vision (ICIEV), Dhaka, Bangladesh*, pp. 1-6, 23-24 May 2014. Article (CrossRef Link)

[11] X.-Y. Sun and W.-Y. Zeng, "Research on sequence of function calls based on gprof," *Microcomputer Information*, vol. 26, no. 36, pp. 165-166, 2010. Article (CrossRef Link)

[12] M. Chabbi, X. Liu, and J. Mellor-Crummey, "Call paths for pin tools," in *Proc. of Annual IEEE/ACM International Symposium on Code Generation and Optimization, Orlando, FL, USA*, pp. 76-86, 15-19 February 2014. Article (CrossRef Link)

[13] A. Feldthaus, M. Schafer, M. Sridharan, J. Dolby, and F. Tip, "Efficient construction of approximate call graphs for Javascript IDE services," in *Proc. of 2013 35th International Conference on Software Engineering, CA, USA*, pp. 752-761, San Francisco, 18-26 May 2013. Article (CrossRef Link)

[14] F. Zyulkyarov, T. Harris, O. S. Unsal, A. Cristal, and M. Valero, "Debugging programs that use atomic blocks and transactional memory," *ACM Sigplan Notices*, vol. 45, no. 5, pp. 57-66, 2010. Article (CrossRef Link)

[15] SystemTap, "Systemtap wiki," 2019. Article (CrossRef Link)

[16] N. A. Carvalho and J. Pereira, "Measuring software systems scalability for proactive data center management," in *Proc. of On the Move to Meaningful Internet Systems: OTM 2010, Hersonissos, Crete, Greece*, pp. 829-842, 25-29 October, 2010. Article (CrossRef Link)

[17] D. de Oliveira and R. S. de Oliveira, "Comparative analysis of trace tools for real-time Linux," *IEEE Latin America Transactions*, vol. 12, no. 6, pp. 1134-1140, 2014. Article (CrossRef Link)

[18] R. Jalan and A. Kejariwal, "Trin-trin: Who's calling? A pin-based dynamic call graph extraction framework," *International Journal of Parallel Programming*, vol. 40, no. 4, pp. 410-442, 2012. Article (CrossRef Link)

[19] O. Levi, "Pin - A dynamic binary instrumentation tool," 2018. Article (CrossRef Link)

[20] B. Gregg, "Linux performance profiling tool perf," 2018. Article (CrossRef Link)

[21] R. Di Pietro and F. Lombardi, "Virtualization technologies and cloud security: Advantages, issues, and perspectives," *From Database to Cyber Security, Springer, Cham*, pp. 166-185, 2018. Article (CrossRef Link)

[22] A. Damodaran, F. Di Troia, C. A. Visaggio, T. H. Austin, and M. Stamp, "A comparison of static, dynamic, and hybrid analysis for malware detection," *Journal of Computer Virology and Hacking Techniques*, vol. 13, no. 1, pp. 1-12, 2017. Article (CrossRef Link)

[23] T.Y. Win, H. Tianfield, and Q. Mair, "Big data based security analytics for protecting virtualized infrastructures in cloud computing," *IEEE Transactions on Big Data*, vol. 4, no. 1, pp. 11-25, 2017. Article (CrossRef Link)

[24] R. K. Barik, R. K.Lenka, K. R. Rao, and D. Ghose, "Performance analysis of virtual machines and containers in cloud computing," in *Proc. of 2016 International Conference on Computing, Communication and Automation, Noida, India*, pp. 1204-1210, 29-30 April, 2016. Article (CrossRef Link)

[25] C.-T. Yang, J.-C. Liu, C.-H. Hsu, and W.-L. Chou, "On improvement of cloud virtual machine availability with virtualization fault tolerance mechanism," *Journal of Supercomputing*, vol. 69, no. 3, pp. 1103-1122, 2014. Article (CrossRef Link)

[26] S. Ribić and A. Salihbegović, "Tiny operating system kernel for education purposes," in *Proc. of 38th International Convention on Information and Communication Technology, Electronics and Microelectronics, Opatija, Croatia*, pp. 700-705, 25-29 May 2015. Article (CrossRef Link)

Longlong Jiao received a Master's degree from the School of Information and Electronics, Beijing Institute of Technology, Beijing, China, in 2013. He is currently pursuing a Ph.D. at the Information System and Security & Countermeasures Experimental Center, Beijing Institute of Technology. His current research interests are computer security and malware detection.

Senlin Luo received B.E. and M.E. degrees from the College of Electrical and Electronic Engineering, Harbin University of Science and Technology, Harbin, China, in 1992 and 1995, respectively, and a Ph.D. from the School of Information and Electronics, Beijing Institute of Technology, Beijing, China, in 1998. He is currently a Deputy Director, Laboratory Director, and Professor of Information System and Security & Countermeasures Experimental Center, Beijing Institute of Technology. His current research interests include machine learning, medical data mining, and information security.

Wangtong Liu received a Bachelor's degree from the School of Information and Electronics, Beijing Institute of Technology, Beijing, China, in 2013. He is currently pursuing a Ph.D. at the Information System and Security & Countermeasures Experimental Center, Beijing Institute of Technology. His current research interests include operating system security and virtualization security.

Limin Pan received B.E. and M.E. degrees from the College of Electrical and Electronic Engineering, Harbin University of Science and Technology, Harbin, China. She is currently working at the Information System and Security & Countermeasures Experimental Center, Beijing Institute of Technology. Her current research interests include machine learning, medical data mining, and information security.