Hindawi
Security and Communication Networks
Volume 2020, Article ID 8702017, 19 pages
https://doi.org/10.1155/2020/8702017

Research Article

BAHK: Flexible Automated Binary Analysis Method with the Assistance of Hardware and System Kernel

Jiaye Pan, Yi Zhuang, and Binglin Sun

College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 200016, China

Correspondence should be addressed to Yi Zhuang; zy16@nuaa.edu.cn

Received 12 December 2018; Revised 13 December 2019; Accepted 20 December 2019; Published 16 January 2020

Academic Editor: Kaitai Liang

Copyright © 2020 Jiaye Pan et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

To protect core functions, applications often utilize the countermeasure techniques such as antidebugging to avoid analysis by outsiders, especially the malware. Dynamic binary instrumentation is commonly used in the analysis of binary programs. However, it can be easily detected and has stability and applicability problems as it involves program rewriting and just-in-time compilation. This paper proposes a new lightweight analysis method for binary programs with the assistance of hardware features and the operating system kernel, named BAHK, which can automatically analyze the target program by stealth and has wide applicability. With the support of underlying infrastructures, this paper designs several optimization strategies and specific analysis approaches at instruction level to reduce the impact of fine-grained analysis on the performance of target program so that it can be well applied in practice. The experimental results show that the proposed method has good stealthiness, low memory consumption, and positive user experience. In some cases, it shows better analysis performance than the traditional dynamic binary instrumentation method. Finally, the real case studies further show its feasibility and effectiveness.

1. Introduction

Most of the software that reaches users is delivered in binary form. To protect intellectual property, many applications use multiple protection mechanisms to avoid external reverse analysis and leakage of implementation details. For malicious programs, it is also important to adopt detection techniques and other countermeasures to impede or mitigate the recognition of malicious behaviors [1–3].

In the field of binary analysis, dynamic analysis can obtain the real execution path of the target program compared with static analysis, but it will face the challenge of runtime countermeasures, especially those advanced malicious codes with kernel modules [4]. Although fined-grained analysis based on the dynamic binary instrumentation (DBI) method may cause severe performance degradation of the analyzing program, optimized methods and implementation could still make it applicable to practical program analysis. There are many representative sophisticated tools with analysis granularity at different levels such as Valgrind [5], DynamoRIO [6], Intel Pin [7], and Frida [8]. For instruction-level analysis, because the instrumentation process necessarily involves code rewriting and interpretive execution, it will lead to the expansion of the original code and the modification of the target program, which are the obvious features that can be easily detected [9]. Similarly, for function-level analysis, hijacking of the function call also requires modification of the target binary program. At the same time, there are additional modules and code related to the instrumentation tool in the process space of target program when it is running. These salient features may make binary instrumentation easier to be detected and attacked [10]. There are other automated analysis methods that are mainly implemented as plug-ins. In essence, they are still based on program debugging and execution emulation methods, such as auxiliary scripts of *WinDbg* and *IDA Pro*. Moreover, it is difficult for such method to achieve automatic analysis of the entire program, and the analysis efficiency is low.

In addition, some research works based on virtual machines (VM) or emulators to implement system-wide application analysis will also face the challenge of many types

of detection methods [11,12], and the virtual machines themselves will cause difficulties in deployment and analysis efficiency. In recent years, due to the development of hardware technology, many studies have begun to take advantage of hardware features, especially hardware virtualization technology, which has been widely used in cloud computing, access control, vulnerability discovery, and program analysis [13–15]. Since the confrontation between analysis and antianalysis exists not only at the kernel level but also in virtualized environments and simulators [16], some studies use more hardware features to further improve the stealthiness of debugging, such as MALT [17] and SPIDER [18]. Based on that, this paper proposes a new lightweight automated binary analysis method. It is named BAHK, which utilizes the new features introduced by Intel virtual machine extensions (VMX) technology. It uses the extended page-table (EPT) mechanism to achieve real-time interception of program execution [19] and integrates EPT violations and virtualization exception (#VE) to further reduce the performance overhead caused by the VMX transitions. Meanwhile, it integrates the hypervisor with the operating system (OS) kernel, which can conveniently obtain the context information of target program. On the basis of program execution interception, a real-time instruction-level automatic analysis is performed. At the same time, three different specific analysis approaches are proposed to meet the needs of different analysis scenarios; thus the proposed method can be used to perform fine-grained dynamic data taint analysis on the target program. Currently, the method proposed in this paper is mainly oriented to the analysis of the program in user mode but can also be extended to add support for the code in kernel mode.

Compared with other methods based on hardware features, the proposed method is aimed at the lightweight automated analysis, based on the instruction level, and oriented to the entire program. The hypervisor is combined with the system kernel to accurately obtain the semantics information and improve analysis efficiency; the combination of multiple optimization technologies can eliminate performance bottlenecks of the virtualization transitions. Compared with the instrumentation methods, the proposed method in this paper has higher stealthiness and stability and can complete the analysis in the environment where some instrumentation methods are difficult to apply. In certain cases, it can achieve better analysis efficiency. In addition, in this method, the analysis and the execution of target program are tightly coupled, which is conducive to controlling and hijacking the execution of target program in time.

In summary, this paper makes the following contributions.

This paper proposes a new method for automated analysis of binary programs. It implements instruction-level analysis of the entire program with the assistance of hardware features and system kernel, which can conduct the analysis of whole program at instruction level. It also designs the interception framework without any modification of the program and has good applicability and stealthiness. Meanwhile, optimization strategies are designed to reduce the additional performance overhead caused by virtualization transitions.

On the basis of interception framework, three specific fine-grained analysis approaches are proposed to support dynamic taint analysis, which can be applied in different analysis scenarios. The first two can perform the real-time online analysis on the target program according to the requirement, while the latter allows offline analysis based on plenty of runtime information recorded online.

The prototype of the proposed method is implemented on Windows platform, and various real experiments are conducted to evaluate the functionality and performance. The experimental results show that the proposed method can be applied to the practical analysis of binary programs with good user experiences.

The rest of the paper is organized as follows. The related works are reviewed in Section 2. Section 3 describes the proposed method and analysis approaches in detail. Section 4 shows the key points of implementation. The conducted experiments and results are demonstrated in Section 5. Section 6 gives the discussion, and Section 7 concludes the work.

2. Related Work

In the past two decades, there existed a lot of research works about binary analysis based on DBI methods, which have achieved significant results [20–22]. Some representative studies are discussed as follows. TaintEraser can detect the data leakage in the commodity software [23], which adopts a series of optimizations to reduce the performance overhead. Kemerlis et al. proposed and implemented the more generalized analysis tool libdft [24], which provides the programming interface and thus facilitates the building of data flow analysis tools. In the latest researches, Jee et al. proposed the accelerated analysis method ShadowReplica [25]. It decouples the analysis from the program execution and then takes full advantage of spare processor cores to perform parallel analysis. Ming et al. proposed the pipelined symbolic analysis method TaintPipe [22]. On the basis of decoupling techniques, it constructs the straight-line instructions by the lightweight instrumentation and accelerates the symbolic analysis by multiple worker threads. After that, the similar offline scheme StraightTaint is also proposed [26], and Wang et al. also proposed another offline analysis method [27]. Recently, Banerjee et al. proposed the new analysis tool Iodine [28], which can avoid the frequent rollbacks in the optimistic dynamic analysis. Most of the researches were based on dynamic binary instrumentation methods that are easily to be detected by the program being analyzed and may also have applicability problems, especially the fact that the target program is graphically intensive. Unlike those research works above, this paper mainly proposes a new interception framework for program execution instead of traditional binary instrumentation and proposes the supporting fine-grained analysis approaches. Some ideas for specific analysis could also be integrated into the proposed analysis framework.

Furthermore, some research works focus on the system-wide analysis in order to obtain the effects on the whole system. For instance, the analysis framework DECAF [12] and its improved version DECAF++ [29] can perform the bit-level taint analysis on the system and provide the developing interface for plug-ins. Lengyel et al. proposed the malware dynamic analysis system DRAKVUF [30]; the virtualization technology is used to monitor the activities in the system kernel. It can detect the behaviors of malware by stealth but mainly captures the behaviors at the function level. Ji et al. proposed the cross-host data flow tracking system RTAG based on the data recording and replaying techniques [31]. Mostly the system-wide analysis is heavily built on the emulators, such as QEMU [32] and Xen [33], and it makes great efforts to acquire the context and semantics information. It usually leads to the costly deployment and the low analysis efficiency to some extent; applicability problems may also exist for the analysis of running programs.

With the rapid development of hardware technologies, more and more researches are integrated with the hardware features [17, 34]. For example, Pan et al. proposed the virtualization-based tool Digtool [13]; it utilizes the shadow page-table and fuzzing technology for detecting kernel vulnerabilities. Lin et al. adopted the virtual clock to counter the timing detection by malware in the dynamic analysis [35]. In the field of binary analysis and instrumentation, Bungale et al. extended Pin tool upon Xen and made it support dynamic instrumentation in the system-wide analysis. Similarly, Zeng et al. proposed PEMU based on QEMU [11], which is compatible with Pin but is also based on binary rewriting techniques. Some studies also utilized the hardware feature to trace the execution of binary program. Willems et al. used branch tracing (BT) feature to log the execution trace [36] in order to perform further offline analysis. The performance monitoring counter (PMC) mechanism is also used to monitor the execution of target program [37]; similarly Bahador et al. tried to detect the malicious behaviors based on PMC [38], but it is hard to obtain the high-level program semantics. Similarly, Dinaburg et al. proposed the malware analysis framework Ether [14], which can trace the instruction execution, memory writes, and system calls. Zhan et al. proposed the solution of kernel control flow integrity based on the virtualization [39], but it provided protections on the page-level check. In addition, Deng et al. proposed the more stealthy binary instrumentation framework SPIDER [18], which can trap the target program by invisible breakpoints. Differently, Zhang et al. proposed the debugging method through system management mode (SMM) of modern CPU [17], which can further improve the debugging transparency. Also Ning and Zhang improved the trace transparency of target program based on TrustZone of ARM [40]. However, the debugging techniques are often associated with the low level of automation and coarse-grained analysis. These researches with the assistant of hardware can make the analysis more transparent and also improve the applicability. However, most of the researches concentrate on the stealthiness but pay little attention to the practicality and automation and often perform the coarse-grained analysis. If we want to automatically analyze the heavy code in real time, the method only depending on hardware features may meet the performance problems or even fail. This paper proposes the lightweight fine-grained analysis framework that can be more readily deployed. It gives consideration to the analysis efficiency, stealthiness, and semantics acquisition at the same time.

3. Method Description

The basic principle is to ensure the correct native execution of target program and reduce the performance impact. The method does not involve the modification of the target program and intercepts the program execution through the EPT violations of processors in the virtualization environment. On this basis, all the instructions that have been executed are then analyzed in time. Since no debugging interface is involved, the impact of antidebugging techniques from user mode is greatly reduced. When the analysis environment is virtualized, the original operating system will be treated as the guest software, and physical memory of the host machine is also virtualized if EPT mechanism is enabled. In this situation, only guest-physical addresses are translated to real physical addresses by traversing the EPT paging structures and the memory management of guest operating system and virtual address space of target program will not be affected. Moreover, the analysis code and hypervisor are implemented in a preloaded kernel module; it directly virtualizes the current environment to make the original OS run as the guest and starts the analysis process, which improves flexibility and deployment of the method.

The EPT mechanism is introduced to support the virtualization of physical memory, which is included in the recent Intel processors [41]; similarly, AMD has the rapid virtualization indexing (RVI) technology [42]. It simplifies the translation from guest-physical addresses to physical addresses instead of the traditional shadow page-table scheme [33]. In addition, the guest-physical address translations can also be cached in translation look-aside buffer (TLB) to improve the speed by which VM accesses the physical memory. Each page-table entry (PTE) of EPT can map the minimum physical memory of size 4 kB. The lowest 3 bits of each entry indicate whether the reads, writes, and executes are allowed from the 4 kB page referenced by this entry. Then, on this basis, it can be implemented for intercepting the data accesses and instruction fetches of target program at the page size granularity. Further, since the Intel sixth-generation microarchitecture Skylake, EPT mechanism supports the EPTP-switching function [19], which allows the program running in VM to change the EPT pointer (EPTP) of current processor to a value chosen from the predefined EPTP list without inducing any VM exit. The size of EPTP list is 4 kB, which contains 512 entries and each entry points to an EPT table. Therefore, with the help of these hardware features, we can design the efficient interception scheme for the program execution.

The architecture overview of BAHK is shown in Figure 1; the core modules mainly involve the kernel of guest OS and

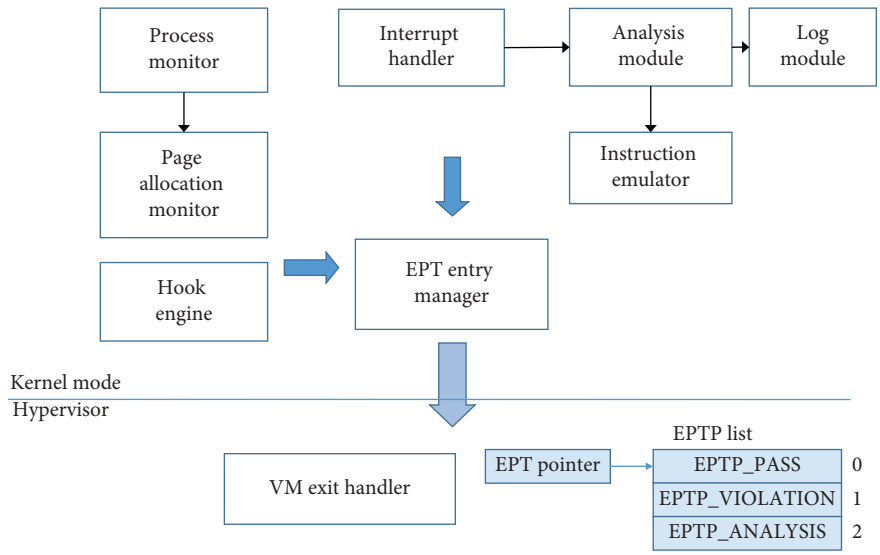


Figure 1: Architecture of BAHK.

hypervisor component. Besides, the configuration and management modules in user mode of guest OS are not shown in the figure. The main analysis task is performed in kernel of guest OS, and the analysis of kernel code is not covered in this paper. Moreover, the hypervisor maintains the EPT paging structures used for the physical address translation and handles the inevitable VM exits. The frequent VM exits will cause the drastic performance degradation of guest OS, so it is very necessary to decrease the number of VM exits.

3.1. Hypervisor Component. Hypervisor is the component that runs between the guest virtual machines and hardware. It is responsible for the mapping of physical resources to virtual ones, also known as virtual machine monitor (VMM). The hypervisor is first loaded as a kernel module of the original operating system and then switches each processor to run in virtualized mode. After that, the original OS will run as the guest VM on logical (virtual) processors, and it will be managed by the hypervisor. The hypervisor only includes basic necessary components of the virtualization environment, such as initialization of virtual machine control data structures (VMCS), allocation and initialization of paging structures for each EPT, transitions between different VMX operations, and handler of VM exits.

In the initialization of hypervisor, it needs to establish the EPT paging structures for each EPTP value that will be shared by all processors and allocate the auxiliary analysis buffer for each processor independently. In the initial situation, the guest-physical addresses are consistent with the physical addresses. As shown in Figure 1, there are three entries contained in the EPTP list, where all memory accesses are allowed in *EPTP_PASS*, *EPTP_VIOLATION* is used for the execution interception of the analyzing program, and *EPTP_ANALYSIS* is loaded in the specific analysis process. It should be noticed that the EPT paging structures consume physical memory resources as well; EPT mechanism uses a page-walk of length 4 to translate low 48 bits of the guest-physical address. For example, if the memory size of host machine is 16 GB, then the full hierarchical EPT paging structures will use about 32 MB physical memory. Therefore, several preallocated EPTP entries do not impact the normal running of guest OS.

The VM exit handler mainly deals with inevitable VM exit events, such as the execution of *cpuid*. In addition, it can aid in confronting the detecting techniques of virtualization in order to improve the stealthiness of modules in guest OS, for example, hiding the hook module.

3.2. Interrupt Handling and Execution Control. The combinations of access control bits in the related PTE of EPT are used to intercept the execution of target program. It mainly focused on access interception of memory reads and writes, assisted by interception of some instruction executes. With the support of #VE, the EPT violation caused by the guest-physical memory access of the guest OS can be converted into an interrupt handled in the guest OS without causing a VM exit event handled in the hypervisor. As shown in Figure 1, the interrupt processing module is responsible for this. Moreover, additional interruption processing is required for debug exception (#DB) and page fault (#PF) interrupts to ensure the analysis continuity by addressing some special analysis exceptions. The analysis module, instruction emulator, and log module coordinate with each other in order to accomplish the automated analysis of the target program. The EPT manager located in kernel of guest OS is responsible for setting access control bits of page-table entries of EPT, switching the EPTP, and synchronizing the EPT paging structures between different processors.

More detailed interception workflow is shown in Figure 2. At the beginning of analysis, the EPTP of each processor points to *EPTP_VIOLATION*, in which access control bits of corresponding page-table entries associated with guest-physical addresses of target program are cleared;

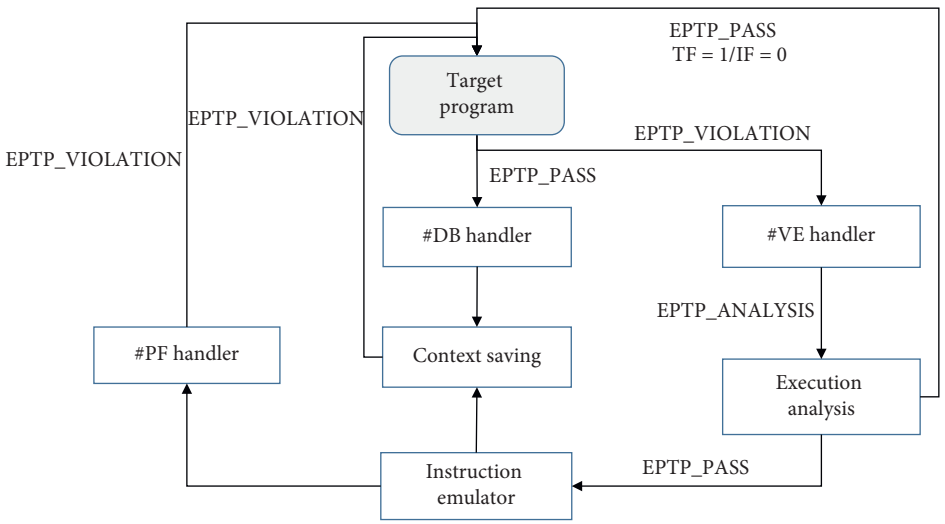


Figure 2: Workflow of the interception for target program.

that means the memory access using the guest-physical address is disallowed and causes an EPT violation accordingly. Currently, only target addresses related to the use mode of guest OS are considered. Therefore, when the target program reads or writes the memory, #VE exception will occur in the guest OS as discussed above. In the handling process of #VE, the EPTP of current processor is switched to *EPTP_ANALYIS* first and then the analysis of instructions that have been executed between the last two #VE exceptions for the same thread of target program will be performed; meanwhile the context of current thread is saved for the next analysis.

When this analysis completes, the instruction that causes the violation needs to be reexecuted. There are two cases for this. One is for simple instructions, including those with *REP* prefix, such as *mov*, *push*, and *pop*. The original memory access can be emulated in the kernel of guest OS but on the premise of security checking. For the complicated instructions, the TF bit of EFLAGS is set in order to make the instruction be executed in single step; then the execution of target thread is subsequently trapped and intercepted in the #DB handler. In the reexecution, the EPTP of current processor points to *EPTP_PASS* and is changed to *EPTP_VIOLATION* after that; thus the execution of target program can be continuously controlled. For single step, it should be noticed that the other interrupts may cause the thread switch of current processor, which will lead to the loss control of interception of the analyzing program. Therefore, the IF bit should also be cleared. In fact, as an alternative approach, we can emulate all memory operations without the single step exception to avoid the temporary mask of interrupt, but this will introduce a small amount of performance overhead due to the context switch while emulating. It can be determined according to the actual requirement. In addition, the #PF exception may occur in the process of emulation, for the reason that the instruction has two memory operands, one of which has not been mapped to the guest-physical address, such as *movsd*. To address this, when #PF occurs in the emulation, the instruction will be reexecuted again by the program. After all the guest-physical pages are allocated, the memory access of that instruction will cause EPT violation again. Then the emulation completes and the analysis continues.

As known from the description above, the interception mechanism will not lose control of the analysis for the running lifecycle of target program, which can obtain the runtime information in time. Therefore, it can perform data flow analysis on the entire program at instruction level. Moreover, because memory accesses are intercepted and analyzed instantly, the consistency of execution and analysis between different threads can be ensured.

3.3. Program Context Management. The process monitor module runs in the kernel of guest OS, which obtains the creation and exit information of processes and threads in real time, also including the image loading and unloading. It allocates the auxiliary buffer associated with the new created thread, which is used to store the context and analysis state when each #VE is being handled. The process monitor also controls the analysis starting and stopping of target program. At the beginning, it suspends the target process and traverses its paging structures in the guest OS to obtain all the allocated guest-physical pages. Then it clears access control bits in the corresponding EPT PTE, resumes the execution of target process, and starts the subsequent interception and analysis.

Page allocation monitor dynamically traces the allocation and release of guest-physical addresses of the analyzing program and modifies the corresponding PTE of EPT in real time. The page allocation module is built on the hook of related kernel functions of guest OS, for the sake of decreasing the number of EPT violations. Besides, the module monitors the variation of guest-physical addresses, but not guest virtual addresses, because the guest-physical address is allocated on demand, which lags behind the virtual address. Since here we focus on the analysis of code in user mode and

do not intercept memory accesses to kernel addresses of target program, actually kernel pages are shared by all processes. Even so, the system kernel and some privileged processes will also access the paging structures of the target process in guest OS, especially in the initialization of target process, but that only causes a small amount of EPT violations.

The monitor modules depend on the hook of related kernel functions; hook engine can use the page redirection technique supported by EPT to improve the transparency. In that case, the redirected physical page does not allow the memory access from the guest OS. If it occurs, a violation will be caused and a deception result will be returned. In addition, the monitor modules also intercept some necessary system services of guest OS to obtain the semantic information and control the analysis progress of target program.

3.4. Execution Analysis. To analyze the target program dynamically at instruction level, it is necessary to obtain all the execution information and track the propagation of data flow. As aforementioned, the EPTP of current processor will be switched to *EPTP_ANALYSIS* in the specific analysis. Some definitions are as follows.

Definition 1. Basic Block. A basic block is a sequence of instructions in a program. Any execution of the sequence starts from the first instruction and exits at the last. Only the last instruction is the branch instruction. For the basic block $B = \{c_0, c_1, \ldots, c_{k-1}\}$, $k > 0$, it contains k instructions, where c_{k-1} is the branch instruction and the execution of c_{s+1} depends on c_s, $0 \le s < k-1$.

Definition 2. Control Flow Graph. Control flow graph $G = (V, E)$ is a directed graph, where the node set V consists of all basic blocks and the edge set $E \subseteq V \times V$. For $B_s, B_t \in E$, it denotes that the execution can arrive at B_t when the last instruction of B_s has been executed. B_s is the predecessor of B_t, and B_t is the successor of B_s. The set of all predecessors of B_t is denoted as $\mathrm{Pred}(B_t)$, and the set of all successors of B_s is denoted as $\mathrm{Succ}(B_s)$.

Definition 3. Execution Path. An execution path is a finite nonempty sequence of basic blocks, which reflects the possible sequence of instructions that have been executed by a program. For path $P = \{B_0, B_1, \ldots, B_k\}$, $k > 0$, its length is k, and for $0 \le i < k$, $B_i, B_{i+1} \in E$. If there are no duplicated basic blocks, it is called a simple path, which means it does not have cycles.

Definition 4. Fault Sequence. It is an instruction sequence that contains the consecutive EPT violations generated during the execution of the same thread. If path $P = (B_0, B_1, \ldots, B_n)$, $n \ge 0$ generates the fault sequence $F = (f_0, f_1, \ldots, f_m)$, $m \ge 0$, in the execution; then $f_s \in \cup_{t=0}^{n} B_t$, $0 \le s \le m$.

In the specific analysis stage, three fine-grained analysis approaches are proposed according to the interception degree and analysis accuracy of the target program. The first is to intercept all memory reads and writes of the target process in the guest OS. The second is to intercept memory accesses to the guest-physical pages with write permissions, except for pages corresponding to the thread stack space. In the third approach, only memory write operations are intercepted based on the second approach. Note that the memory operations mentioned here refer to addresses in user mode of guest OS. It can be seen that the interception range of the three approaches is reduced in turn, while the corresponding analysis complexity is increased. The main purpose of reducing the interception degree is to reduce the performance impact on performance caused by frequent EPT violations.

3.4.1. Intercepting All Memory Reads and Writes. The specific analysis is performed between the two consecutive EPT violations of the same thread of analyzing program. When all memory reads and writes are intercepted, the instruction that is directly related to the access can be analyzed during the handling of current EPT violation. But, before that, it is necessary to analyze those instructions that have been executed from the last to the current violation of the thread to ensure the analysis continuity. It is assumed that the execution path $P = (B_0, B_1, \ldots, B_n), n \ge 0$, generates the fault sequence $F = (f_0, f_1, \ldots, f_m)$, $m \ge 0$, for the specific thread of target process.

If the instructions that cause the two consecutive EPT violations are the same or consecutive, then there are no additional instructions to be analyzed. Otherwise, if the two instructions are located in the same basic block and the second instruction depends on the first, then we only analyze these instructions between the two. That is, $\exists k, s,\ 0 \le k < m, 0 \le s < n,\ f_k, f_{k+1} \in F$ and $f_k, f_{k+1} \in B_s$; meanwhile $\exists c_i, c_j \in B_s, 0 \le i < j < |B_s|, c_i = f_k, c_j = f_{k+1}$. The $j - i - 1$ instructions between c_i and c_j need to be analyzed first.

If the two instructions are not located in the same basic block, then $\exists k, s, t,\ 0 \le k < m, 0 \le s, t < n, s \ne t$ and $f_k, f_{k+1} \in F,\ f_k \in B_s,\ f_k \in B_s$. If they are in the same block, but the first instruction depends on the second, then $\exists k, s,\ 0 \le k < m, 0 \le s < n,\ f_k, f_{k+1} \in F$ and $f_k, f_{k+1} \in B_s$; meanwhile $c_i, c_j \in B_s, 0 \le i < j < |B_s|,\ c_j = f_k, c_i = f_{k+1}$. For the above cases, we need to know the branching information of execution to obtain the accurate execution path. Although the Intel Processor Trace (IPT) feature can help to get the control flow traces of the target program, the decoding of highly compressed log may produce the additional performance overhead and complex implementation [43]. Therefore, for the online analysis, IPT is not utilized here. Instead, we construct the partial control flow graph for the running program first and determinate the real execution path between the two EPT violations by traversing the graph; then all the instructions executed can be analyzed. The depth first search (DFS) algorithm is used here; the graph building and instruction parsing are simultaneous. The detailed process of execution path extraction is shown in Algorithm 1.

```
Input: Start instruction start_ins, end instruction end_ins
Output: The constructed partial control flow graph and the possible execution paths
(1)  Construct the node of basic block that contains start_ins, and push it into the aided stack stack_dfs.
(2)  while stack_dfs is not empty do
(3)    Get the top node block from stack_dfs.
(4)    if block is parsed then
(5)      if block contains end_ins then
(6)        Save the traversed nodes, and pop block from stack_dfs.
(7)      else if all the adjacent nodes of block are visited then
(8)        Pop block from stack_dfs, mark all adjacent nodes as non-visited.
(9)      else
(10)       Get the adjacent node that is not visited, and push it into stack_dfs.
(11)       Check and mark the loops in the traversed nodes.
(12)     end
(13)   else
(14)     Disassemble and parse block
(15)     while Get instruction ins from block successfully do
(16)       if ins is a system call instruction or has memory operations then
(17)         Mark block as abort
(18)         break
(19)       else if ins is a direct unconditional jump instruction then
(20)         Construct the node of target block, and push it to stack_dfs
(21)         break
(22)       else if ins is a conditional branch instruction then
(23)         Construct all successor nodes of block, and push them to stack_dfs
(24)         break
(25)       else if ins is an indirect branch instruction or other transfers then
(26)         Mark block as pending, and save the traversed nodes and execution paths
(27)         break
(28)       else if ins is the end_ins then
(29)         Mark block as the end
(30)         break
(31)       else
(32)         Save the parse result to block
(33)       end
(34)     end
(35)   end
(36) end
```

Algorithm 1: Algorithm of execution path extraction.

Then the generated execution paths will be refined after extraction completion. Besides, there should be some protection measures for potential exceptions in the algorithm execution; for example, the target code page of conditional jump has not been loaded into memory, and it should be checked first. Furthermore, we can also introduce some optimizations; for example, when the possible execution path is found for the first time, the traverse can return early if its depth is too large. Therefore, the search algorithm can be terminated in good time.

Because all memory accesses of the target process are intercepted, in theory, it is easy to walk to the instruction that causes the current EPT violation from the position where last violation occurs only by the register operations and static analysis. But there still exists some complicated situations; it is considered that several cases of the real execution path are as follows. Assume that the start block is $B_s = \{c_0, c_1, \ldots, c_{p-1}\}$, $p > 0$, the end block is $B_t = \{c'_0, c'_1, \ldots, c'_{q-1}\}$, $q > 0$, and $B_s \neq B_t$, $f_k, f_{k+1} \in F$, where f_k and f_{k+1} are the instructions that cause the consecutive EPT violations.

(1) Single Simple Path. There is only one path from the last to the current violation position in the constructed partial control flow graph, and it is acyclic. Therefore, the path is the real execution path; an example is shown in Figure 3(a). In this case, for the real path $(B_s, \ldots, B_t), t > s$, and $c_i = f_k, c'_j = f_{k+1}, c_i \in B_s, c'_j \in B_t, 0 \leq i < p, 0 \leq j < q$. If $t = s + 1$, then $(B_s \backslash \{c_0, \ldots, c_i\}) \cup (B_t \backslash \{c'_j, \ldots, c'_{q-1}\})$ are the instructions that will be analyzed. Otherwise, if $t > s + 1$, those instructions are included in the set $(B_s \backslash \{c_0, \ldots, c_i\}) \cup (\cup_{l=s+1}^{t-1} B_l) \cup (B_t \backslash \{c'_j, \ldots, c'_{q-1}\})$.

(2) Single Path with Cycle. It means that there are loops in the real execution path; an example is shown in Figure 3(c). Then, for the path $(B_s, \ldots, B_l, \ldots, B_t), s < t$, there exists c, $s \leq c \leq l$, such that $B_c \in \mathrm{Succ}(B_l)$. At this time, according to the analysis requirement, we can reexecute the instruction

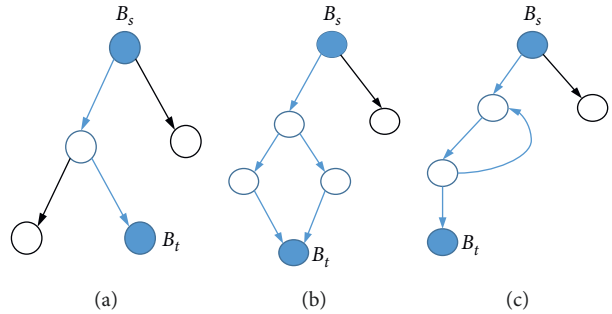


Figure 3: Examples of the special execution paths.

sets $\cup_{i=s}^{l} B_i$ and $\cup_{i=c}^{l} B_i$, respectively, in the separate environment to determine the real number of loops and then unroll the loops to analyze these instructions in order. After that, the analysis continues using the block in the set $\mathrm{Succ}(B_l) \backslash B_c$ as the new start point. The analysis of nodes not in the cycle is the same as the case in the previous paragraph.

(3) Several Simple Paths. There are at least two possible paths in the extracted execution paths, as shown in Figure 3(b). Then we still need to reexecute part of the instructions in the paths to obtain the real execution path. For two possible paths $(B_s, \ldots, B_l, B_{l+1}, \ldots, B_t)$ and $(B_s, \ldots, B_l, B'_{l+1}, \ldots, B_t)$, where $t > s+1, s \le l < t$ and $B_{l+1}, B'_{l+1} \in \mathrm{Succ}(B_l)$, the instruction set $\cup_{i=s}^{l} B_i$ which is composed of the same prefixes of two paths is then executed again to determine the real successive basic block. If the number of extracted paths is more than two, we can determine it step by step with the same method to obtain the final real execution path. For the cyclic path, it can be addressed as the last paragraph.

(4) Other Cases. If no possible paths are found in the partial control flow graph, one case is that it is difficult to construct the partial control flow graph only with static analysis; for example, the search process is aborted by the far transfer instruction *jmp eax*, or the search depth is too large. At this moment, the real execution path will be determined by the partial reexecution similar to precious cases. Besides, the search may encounter system call instructions and abort, which is regarded as the normal termination. It does not matter, because we only analyze the program in user mode. When the system call returns, a timely violation will occur again; then the analysis can remain continuous.

It can be known from discussions above that the real execution path can be directly determined without any execution when there exists only one simple path. In fact, this case will be in the majority for most programs. For other cases, the partial reexecution is needed, which involves mainly the computation between registers, because all memory accesses are intercepted.

3.4.2. Intercepting Memory Reads and Writes except the Stack. The running program will access memory frequently, especially the stack space of thread, which will cause a great amount of EPT access violations. Therefore, the interception degree can be reduced when the performance of running program is expected to be improved. At first, do not intercept the reads and writes to the thread stacks; besides, only these guest-physical pages with write access permissions in guest OS are intercepted. It is because every thread has its own stack, and they do not interfere with each other normally, while the state of read-only pages does not change in the running of the program and permission changes can also be instantly captured. Accesses to these pages that are not intercepted can be timely reproduced during program execution and analysis.

As the reads and writes of some memory pages are not intercepted, it is difficult to directly determine the real execution path through the approach in last section when EPT violations occur. In this case, we use a partial reexecution method to determine the actual execution path of the target thread from the last EPT violation to the current violation, which is also discussed in the last approach. First, after each EPT violation handling is completed, the current thread context information should be saved as well as its allocated stack memory state. Then, when the specific analysis of current thread is performed in the next consecutive EPT violation, the EPTP of current processor is switched to *EPTP_ANALYSIS*. It is worth noting that, in a multicore environment, only the current processor core performs the switching operation, while the other cores remain in their original state. In the paging structures of *EPTP_ANALYSIS*, physical pages corresponding to the current thread stack are mapped to the buffer that stores the stack state in the last EPT violation. As shown in Figure 4, the virtual address of thread stack in guest OS ranges from 0x3F70000 to 0x3F72000, corresponding to two guest-physical pages, whose starting addresses are 0x441CA000 and 0x471C9000, respectively. The guest-physical address is then translated to the host physical address by traversing the four-level mapping structures of EPT [19]. In paging structures of *EPTP_PASS* and *EPTP_VIOLATION*, the guest-physical addresses are consistent with host physical addresses. In *EPTP_ANALYSIS*, the translated addresses are mapped to a preallocated continuous buffer, which holds the stack status of current thread when the EPT violation was last generated. At this point, we can reexecute these instructions from the last EPT violation instruction of the current thread until the current EPT violation instruction, while using the context saved in the last violation as the initial thread context in the reexecution. Then the real execution path can be reproduced

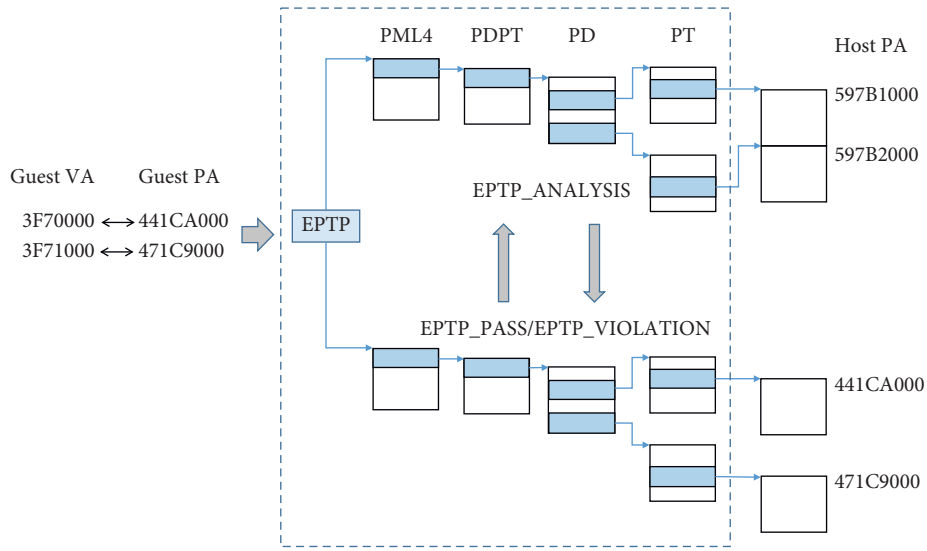


Figure 4: Example of the guest-physical address translation through different EPT paging structures.

and analyzed. In addition, during the reexecution, data flow analysis at instruction level can be synchronously performed, and the instant context of the thread before each instruction can also be obtained.

In the formalized description, it is also assumed that the execution path of specific thread is $P = (B_0, B_1, \ldots, B_n), n \geq 0$, which generates the fault sequence $F = (f_0, f_1, \ldots, f_m), m \geq 0$. Similar to the last section, if the instructions that cause consecutive EPT violations are the same or consecutive, the analysis of other instructions is not needed. If the instructions are located in the same block and the second instruction depends on the first, that is, $\exists k, s,\ 0 \leq k < m, 0 \leq s < n,\ f_k, f_{k+1} \in F,\ f_k, f_{k+1} \in B_s$, and, for $c_i, c_j \in B_s, 0 \leq i < j < |B_s|$, there exists $c_i = f_k, c_j = f_{k+1}$, then $j - i - 1$ instructions between c_i and c_j will be analyzed. But, differently, there may be some instructions between c_i and c_j which have memory accesses to the stack space. Therefore, the partial reexecution is also needed. More specifically, it saves current context of the thread first and then loads the last saved context into the processor and the stack pointer register is also switched. Due to the elaborate assembler code, the original context is automatically restored when the reexecution completes.

Otherwise, for $f_k, f_{k+1} \in F, f_k \in B_s, f_{k+1} \in B_t$, $0 \leq k < m, 0 \leq s < t < n$, where f_k and f_{k+1} are the instructions that cause the consecutive violations, the reexecution and analysis will start at f_k until f_{k+1}. If the current basic block is $B_l, s \leq l < t$, then the next block B_{l+1} is determined by instruction set $\cup_{i=s}^{l} B_i$. In the partial reexecution, the parsed instructions are cached in the auxiliary buffer. If the transfer target of current block can be determined without runtime information, then the instructions $B_l \cup B_{l+1}$ are continuously cached until the transfer target cannot be determined by static analysis. At this moment, we switch the context of the thread and execute the cached instructions to obtain the address of next basic block.

In the normal condition, the real execution path $(B_s, \ldots, B_l, B_{l+1} \ldots, B_t)$ can be successfully reproduced and analyzed, and basic blocks in set $\cup_{i=s}^{t-1} (\mathrm{Succ}(B_i) \backslash B_{i+1})$ are not involved. However, there are lots of memory accesses and context switches in the process of reexecution; it is necessary to design fault tolerance schemes to avoid the analysis getting out of control. Two possible cases may occur. One is that the obtained sequence of basic blocks is different from the real one, but the last basic block is correct. It means that the reexecution can terminate normally, which does not impact the running of target program. At this point, the obtained sequence is like $(B_s, \ldots, B_l, B'_{l+1}, \ldots, B_t)$. In the other case, the partial reexecution may generate the sequence $(B_s, \ldots, B_l, B'_{l+1}, B_l, B'_{l+1}, B_l, \ldots)$, so that it will not terminate by itself. For this case, we can set a threshold for the length of obtained sequence, which makes the reexecution be able to terminate. Besides, some special instructions should be separately addressed, such as *popfd*. Moreover, because the partial reexecution does not involve accesses to the memory outside the stack space of current thread, we can intercept these memory accesses by modifying the corresponding PTEs in *EPTP_ANALYSIS* to detect the out-of-bounds errors. For other errors that cannot be discovered online, IPT can also be used to log the full control flow traces, and then the analysis can be verified after the execution of target program. Moreover, the stack space of current thread may be affected by another thread; then the specific instructions that cause the effects can also be located and addressed additionally.

More detailed algorithm of execution path analysis is shown in Algorithm 2. In addition, the procedure of context switching includes the saving of current context and loading of last saved context of the thread. The transfer target of some branch instructions can be inferred from the assembly of target program, for example, near relative *jmp* and *call*. For the instruction *ret*, indirect call, and so on, the transfer target cannot be calculated without execution. Moreover, the EPTP of current processor should also be switched before and after the algorithm execution.

Input: Start instruction *start_ins*, target instruction *end_ins*
Output: Execution path and analysis results
(1) Initialize the auxiliary buffer *buf* for current processor and thread.
(2) *curr_ins* ⟵ *start_ins*
(3) while *curr_ins* is not equal to *end_ins* do
(4) Parse *curr_ins*.
(5) if *curr_ins* is a branch instruction then
(6) if the transfer target can be directly determined then
(7) Get the target instruction *target_ins*.
(8) Insert *curr_ins* and extra analysis code to *buf*.
(9) *curr_ins* ⟵ *target_ins*
(10) break
(11) else if the transfer target cannot be directly determined then
(12) Switch the context, execute instructions of *buf* and perform analysis.
(13) Get the jump target address *target_ins*.
(14) *curr_ins* ⟵ *target_ins*
(15) break
(16) else if *curr_ins* is a system call, interrupt or trap instruction then
(17) break
(18) end
(19) else
(20) Insert *curr_ins* and extra analysis code to *buf*
(21) end
(22) end
(23) Switch the context, execute and analyze the remaining instructions of *buf*.
(24) Save the execution path and analysis results.

Algorithm 2: Algorithm of execution path analysis based on partial reexecution.

3.4.3. Intercepting Only Memory Writes except the Stack. This approach shrinks the interception range further, which only intercepts the memory writes of the analyzing program based on the second approach discussed in the last section. On the contemporary computers, most applications are multithreaded; different threads that access the same memory should be synchronized. In this interception scheme, while the program is running, it is difficult to determine the result of memory read operation, because memory reads are not intercepted and may be affected by several threads in a short period. An example is shown in Figure 5; thread *A* causes two EPT violations when writing the memory addresses *Addr_0* and *Addr_2*; instructions executed between the two events will be first analyzed in the handler of the latter violation. However, there is a read access to *Addr_1* which is not intercepted, and thread *B* also writes to the same address many times in actual execution. As a result, it is difficult to confirm the real value that thread *A* read previously from *Addr_1* by the online backtrace analysis.

In this situation, we log the memory write information for each EPT violation and to perform the offline analysis after the program exits. The log is then saved to the disk by the thread outside the target process. When the target process is created, the allocated memory region is also saved. Similarly, the initial state of context and stack of the thread is saved after it is created. Based on this information, the offline analysis can be fairly accurately accomplished. Besides, because the memory write operations that come from the kernel of guest OS are also intercepted and recorded, the memory data produced by the system call will not be lost.

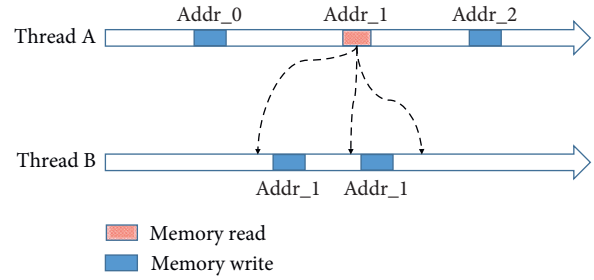


Figure 5: Problem of memory read analysis with multiple threads.

For the further offline analysis, which can refer to the existing research works [26,44], this paper does not focus on this currently. Differently, the control flow information is not recorded here, which can only be determined by the offline emulation. In fact, as previously mentioned, the IPT mechanism can also be adopted. Additionally, the memory access instructions of the thread should be noticed in the offline analysis, as shown as follows:

(i) Address is within the stack region. The operation is directly emulated on the simulated stack of the thread

(ii) Write to the address outside the stack. Because the real write information is logged, then the value to write is compared with the log for the verification. If they are not equal, that means the precious analysis

has errors; otherwise the correct value is written to the simulated memory space

(iii) Read from the address outside the stack. First we obtain the two memory writes before and after this operation of current thread from the log; meanwhile check if there exists a write to this address by another thread during the period of the two writes. If not, the value in the simulated memory address is returned. Otherwise, for each write of other threads, the value to write is returned to the current read, and the analysis of current thread continues. Then, at the next memory write of current thread, the context of analysis is compared with the saved one in the log. If the two are inconsistent, the analysis is rolled back to the state of last read and repeated using another value of write; otherwise the analysis continues. If another memory read of current thread occurs in the trial analysis, it will be handled recursively with the same method

3.4.4. Comparison of Three Specific Analysis Approaches. Based on the execution interception, we propose three specific analysis approaches to perform the instruction-level analysis as described above, which can meet different analysis scenarios. The first two approaches perform the coupled online analysis, which is emphasized here, as it is more accurate and can be rapidly repeated to obtain the analysis result in time.

First, the performance impact on the target program reduces gradually in the description order, since the interception degree of target program decreases. The main measure is to reduce the number of memory accesses that cause EPT violations, so that the runtime performance of target program will be improved accordingly.

Second, the complexity of analysis would increase if these approaches above are adopted in turn; it means that the amount of runtime information obtained in the analysis is different. The latter two do not obtain enough dynamic information, such as some memory addresses with values, thread synchronization, and return values of system calls. So more information needs to be inferred, which makes the analysis more complicated.

Third, the approaches differ in applicability. The first one can finely control the execution of target program and acquire the information of memory accesses and branching in real time. It is widely applicable but would lead to the performance reduction if the target program is memory access intensive; we will discuss this case again in Experiments and Result Analysis. The second approach further increases the analysis' efficiency; it reproduces the prior memory accesses to the stack space of current process based on the premise that each thread has its own private stack space. But, in the extreme case, different threads may influence each other in the stack space, for example, if a thread writes the stack belonging to another thread, which would not be intercepted and may lead to synchronization problems in the subsequent analysis. Fortunately, we can discover the anomaly in time and adjust the analysis strategy, as discussed above. The applicability of the last approach depends on the complexity of offline analysis, which can be a choice if the target program is sensitive to delay. Other offline analysis ideas can also be combined with the runtime recording method here.

Finally, the memory consumption of analysis increases in the description order of three approaches. The first two depend on the size of native code to be analyzed of target program. The needed memory is limited, as we do not cache the analysis code currently. In the second approach, the auxiliary memory space is allocated for each thread, which has the same size as the original stack. Then additional buffers are needed for runtime data recording in the last approach.

In addition, we can set the extent of interception according to the specific analysis requirement. For example, to acquire all the function calls of the running program, we only need to intercept memory writes to the stack space; if our concerns are the data flows in and out of a fixed memory space, then the interception only covers the accesses to that region. Also, we can radically reduce the extent of interception according to the features of target program. For example, accesses to the memory belonging to single thread could be excluded from the interception.

4. Implementation

The prototype is implemented on the 32-bit Windows platform, which has more requirements of malware analysis. Due to closed source, it is suitable for the analysis of inside programs with the help of external artifacts. The implemented analysis framework includes the kernel driver and the console program in user mode. The user module controls the loading of kernel driver and configures the parameters of analysis. The kernel driver contains the hypervisor infrastructure, analysis module, and other assisted modules. The involved key implementations are described as follows.

The hypervisor component is developed on the basis of HyperPlatform [45], most of irrelevant code is removed, and unnecessary VM exits are masked. In the initialization of VMCS, the requisite control fields are set to enable the functionality of virtualization exception, for example, "*Enable VM functions*" and "*EPT-violation #VE.*" Additional buffer is allocated to store the virtualization exception information and then registered in the VMCS. When the EPT paging structures are preallocated, the "*EPT pointer*" and "*EPTP-list address*" controls are also configured to enable EPTP-switching mechanism. Moreover, we add the interrupt service routine for the interrupt vector 20 in the guest OS to handle the EPT violation. Modules of kernel driver in the guest OS interact with the hypervisor by *VMCALL* and *VMRESUME* instructions.

The handlers *KiTrap01* and *KiTrap0E* of the guest OS are intercepted to handle the #DB and #PF exceptions in the analysis. Similarly, the undocumented functions *MiAllocateWsle*, *MiCopyOnWriteEx*, and *MiDeletePteRun* are also hooked to track dynamically the changes of working set of target process and update the access control bits in the corresponding EPT PTE synchronously. The exported

function *KeIpiGenericCall* is used to synchronize the modifications between different processors. Several callback functions are registered by the exported functions *PsSetCreateProcessNotifyRoutineEx* and *PsRemoveCreateThreadNotifyRoutine* to manage target processes and threads.

In the handler of #VE, these violations that are not caused by the analyzing process are ignored, and EPTP switching is done by the *VMFUNC* instruction. *udis86* is used as the lightweight dissemble engine [46], which may not be the best choice. In the processing of single step, TF and IF flag bits are set in company and then restored in the subsequent handler of #DB. Moreover, each processor has the preallocated auxiliary buffer used for partial reexecution in the analysis. Before the reexecution starts, the current context of thread is saved; then the context loading code, target instructions, analysis code, and context restoration code are successively copied to the assisted buffer. After reexecution, the original environment will be naturally restored.

5. Experiments and Result Analysis

The experiments are primarily conducted on the common personal computer, which contains *Intel i5-7500* @ 3.40 GHz 4 cores CPU, 4 GB Memory, 120 GB *Solid State Drive,* and 500 GB (7200 RPM) *Hard Drive*. 32-Bit *Windows 10 (10240)* is installed on the solid state disk. *Intel Pin 3.4 97438, Visual Studio 2015 Enterprise Update 3,* and *Windows Driver Kit 10.0.14393.0* compose the main development environment. First we test and evaluate the performance of many programs with different specific analysis approaches and perform comparison with Pin at the same time. Subsequently, the real case studies are used to verify the feasibility and effectiveness.

5.1. Performance Evaluation. For the first specific analysis approach, which is to intercept all memory accesses of the target program, we select several programs commonly used in the Windows platform as the target for experiments. The specific program list is shown in Figure 6, including built-in applications and some popular third-party applications. In the current test scenario, the built-in *Certutil* was used to calculate the SHA256 hash value of a 1 MB text file. *Notepad* and *XPS* with graphical user interfaces (GUI) were used to test the startup and exit processes. *FFmpeg* is a famous audio and video processing tool [47], which was used to convert a 20K wav file to the mp3 format. Continuously, *Curl* was used to download a 20 MB text file from a local server [48], and the command line provided by *WinRAR* was used to compress a 1 MB text file by default arguments. During the experiment for each test program, *Pin* was first used to instrument the instructions that involve memory accesses and count the number of executions, denoted as PM in Figure 6; then the first analysis approach is performed on the same program. Each procedure was repeated 10 times and the average of the relevant results was calculated. The experimental results are shown in Figure 6 and Algorithm 1. In Figure 6, the time consumption of executing the test scenario without instrumentation by Pin is used as a baseline value to calculate the performance degradation in other cases. In some test scenarios, the time that the program takes to natively execute is short, which leads to statistical errors, and the impact on the target program is relatively limited without instrumentation by Pin [24,26]. Moreover, FI indicates the performance degradation resulting from the interception of memory operations only when the first method was adopted; FA indicates that, based on the interception, all the execution instructions of the target program were analyzed.

As seen from the results in Figure 6, when the running program involves frequent memory accesses such as compression and decoding, the proposed method will cause the great performance degradation of target program. But, for these programs with GUI, it will achieve better performance than *Pin*, which is possibly affected by the handling of a large amount of window messages. Moreover, the extra analysis has lower impact on the performance, compared with the handling of EPT violations, which are the main restraining factors. Then, as seen together with the detailed results in Table 1, simple paths of the first specific analysis approach account for the majority of all possible paths, which could avoid the performance overhead caused by lots of reexecutions. The processor time is higher when the number of threads grows, depending on the implementation of target program as well. Generally, BAHK has less memory consumption than *Pin*, which mainly includes the usage of driver module itself and allocated auxiliary buffers for processors and threads. The analysis is also affected by the number of threads but does not have significant changes. In fact, *Pin* consumes more virtual memory addresses of the process.

In the preceding scenarios, the number of memory accesses is relatively small for most programs in the running lifecycle. That is the most applicable scenario for the proposed method in this paper. Next we observe the changes of performance by two compression programs when the number of EPT violations increases. We use *7-Zip* to compress the plain file to the zip format with different original sizes in the range of 100 kB~1 MB, respectively [49]; then the same compressions are also performed by *WinRAR*. As shown in Figure 7, the denotations of PM and FI are the same as the previous experiment, and then the absolute execution time is used to measure the performance. When the number of EPT violations increases, the performance overhead will present the linear increase, but the rate of growth is not the same for different programs. In addition, the performance impact caused by *Pin* is not obviously augmented; it should be that its cache mechanism helps to reduce the number of instructions to be repeatedly instrumented, which are mainly located in the loop of compression.

In the next experiment *Curl* is used to download the same plain file of size 10 MB with different downloading speeds separately, and the original file has been compressed to the gzip format in the transmission. With this test, we evaluate the performance overhead affected by the network condition. The execution time is shown in Figure 8, where

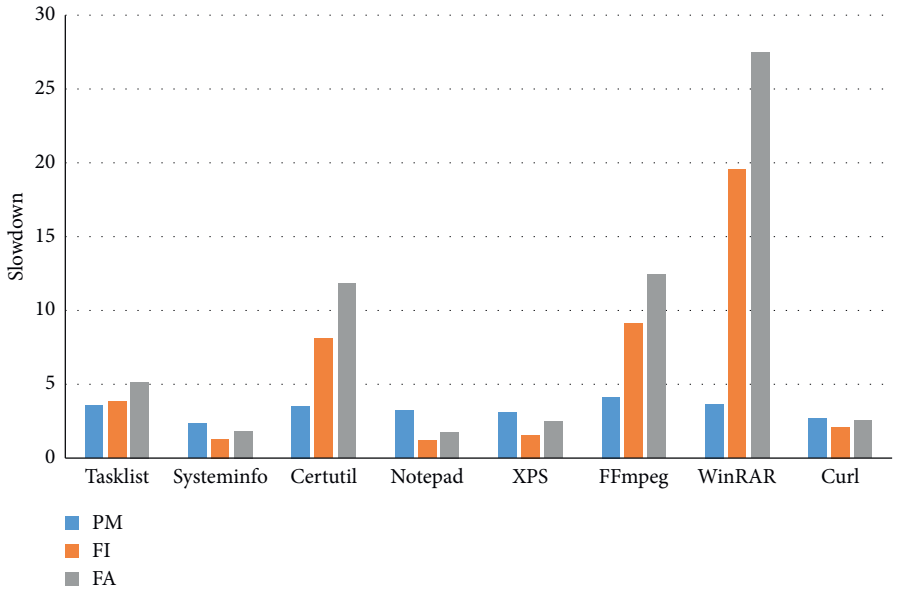


Figure 6: Performance degradation of the first specific analysis method compared with Pin.

Table 1: Other results by the first specific analysis approach.

Program	Number of EPT violations (million)	Proportion of simple paths (%)	Processor time (PM-FI)	Physical memory occupation (PM-FI)	Execution time of PM (s)
Tasklist	16.8	91	24%-24%	53M-41M	10.2
Systeminfo	6.9	88	22%-24%	51M-37M	9.3
Certutil	25.3	92	25%-25%	45M-36M	7.2
Notepad	9.1	87	18%-19%	84M-42M	20
XPS	16.6	89	22%-23%	90M-44M	21
FFmpeg	34.8	91	21%-24%	64M-45M	11.2
WinRAR	201.1	98	28%-65%	92M-85M	9.7
Curl	8.9	94	16%-21%	33M-35M	9.0

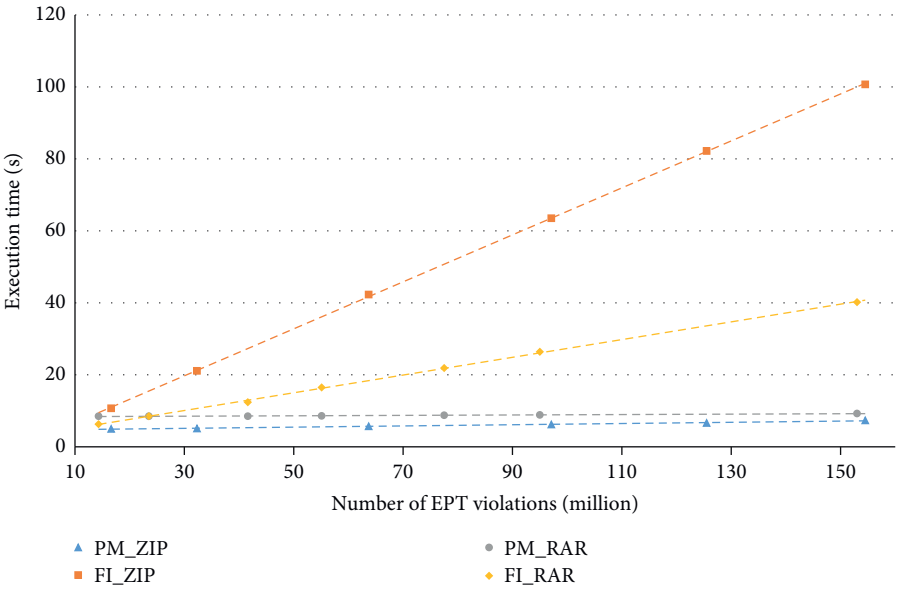


Figure 7: Execution time for different numbers of EPT violations.

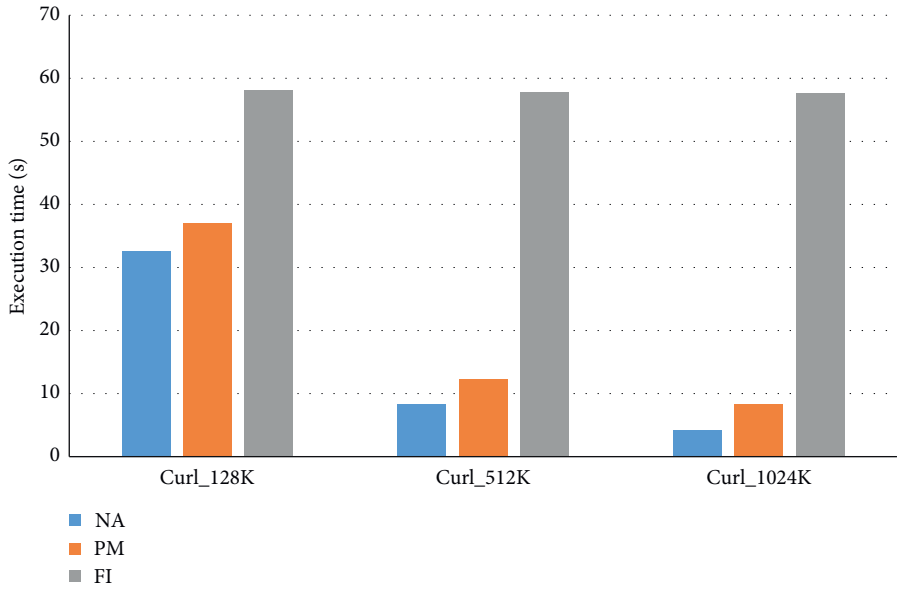


Figure 8: Execution time of Curl with different downloading speed limits.

NA denotes the native execution of the program, and the number of memory accesses is nearly the same, which is about 89 million. When the network speed is reduced, the performance impact also declines. It means that the performance impact actually depends on the occurrence of memory accesses per unit time. For these programs with network accesses, the network delay can mitigate the performance impact to some extent.

Another experiment is to evaluate the performance overhead caused by different generations of CPU. Upon the previous experiment environment, Intel i3-6320T @ 3.40 GHz CPU is added to make comparisons. In the test, *7-Zip* and *WinRAR* are still applied; the former compresses a text file of size 1 MB to 7z format and the latter compresses the same to RAR format by default. We then calculate the average execution time for each of million memory operations. The results are shown in Figure 9. When the performance of CPU is improved, better performance of interception would be achieved. But the handling EPT violations have more performance gains that may be more significant in the latest generation of CPU. Besides, lots of threads run concurrently while *WinRAR* is working. The proposed method prefers the multithreaded program more than Pin does; the same case can also be seen from Figure 6.

In summary, the experimental results above have shown that EPT violation has a significant performance impact on the target program. Therefore, the main purpose of subsequent approaches mentioned in Section 3.4 is to reduce the number of EPT violations. Then we make performance comparisons between the first and second specific analysis approaches. The test program list is shown in Figure 10, similar to the last test, but the test scenarios are different, which will produce more EPT violations in the program execution. In detail, *Certutil* was used to calculate the hash value of the 10 MB text file, and *OpenSSL* was used to encrypt the 5 MB text file with the AES-256-CBC mode [50]; then a WAV file of size 100K was converted to the MP3 format by *FFmpeg*. At last *7-Zip* and *WinRAR* were still used to compress the file of size 1 MB; the target formats were 7z and *RAR*, respectively.

The results are shown in Figure 10 and Table 2. In Figure 10, the ordinate shows the absolute execution time of the test program, while SI denotes that the memory accesses are intercepted by the second specific analysis approach, and SA denotes that all instructions executed are analyzed upon SI. As seen in Figure 10, the second specific analysis approach achieves better performance than the first, for the reason of the reduction of EPT violations. Particularly for *Certutil* and *FFmpeg*, the performance of SI is close to PM, because they involve a great deal of memory accesses to the stack, which can also be verified from Table 2. For the compression programs, the performance impact caused by SI is still significant, but it is lowered on *WinRAR*. While the multithreaded program is analyzed in the coupled manner by the proposed method, we can take full advantage of the processors, which remains consistent with the previous result. Moreover, as known from Table 2, for the second approach, the number of context switches for partial reexecution is small. After all, the reduction of EPT violations is more helpful to the performance improvement.

Finally, several compression programs are used to evaluate the performance impact caused by the third specific analysis approach. It further demonstrates the performance gains due to the reduction of interception degree, according to the analysis strategy. First, the text file of size 5 MB was to be compressed to the gzip, bz2, and 7z format by 7-Zip separately. The test scenarios of *OpenSSL* and *WinRAR* were the same as the last, and another built-in program, *makecab*, was also used to compress the same file. The results are shown in Figure 11, where the absolute execution time in different scenarios is shown on the left ordinate axis; meanwhile the number of EPT violations corresponds to the right ordinate. In addition, FN and TN denote the number of EPT violations by the first and third specific analysis

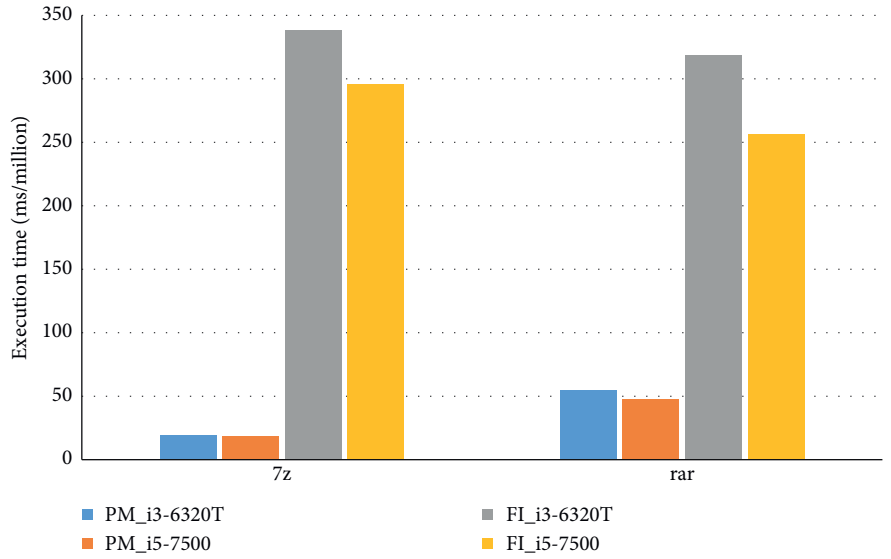


Figure 9: Execution time per million memory accesses for different programs and CPUs.

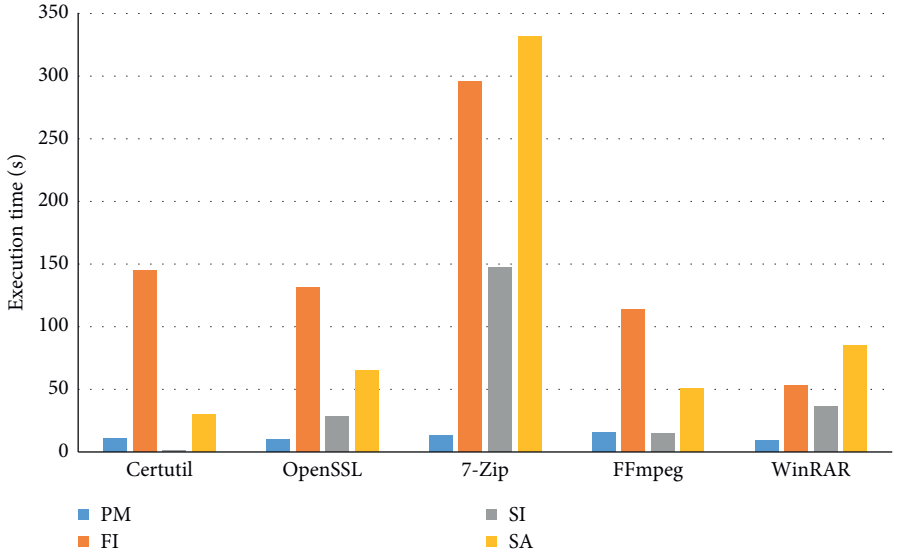


Figure 10: Performance comparisons between the first two specific analysis approaches.

Table 2: Other analysis results by the second specific analysis approach.

Program	Number of EPT violations (million)	Proportion of EPT violations in all memory operations (%)	Number of context switches per violation	Execution time of PM (s)
Certutil	2.6	1.4	1.23	10.5
OpenSSL	30.5	16.6	0.34	9.9
7-Zip	304	42.2	0.42	13.1
FFmpeg	21.2	13.8	0.73	15.3
WinRAR	123.1	61.3	0.26	9.6

approaches, respectively. As seen from Figure 11, compared with the other two approaches, the number of EPT violations decreases obviously because only the memory writes outside stacks are intercepted. When the number of memory accesses exceeds 1 billion, the reduction of EPT violations can avoid the performance drop of target program in the analysis. For different programs, the running performance overhead is mainly related to the absolute number of memory accesses that will be intercepted. Besides, because the offline part does not impact the running of the program, the overhead is not the emphasis here currently.

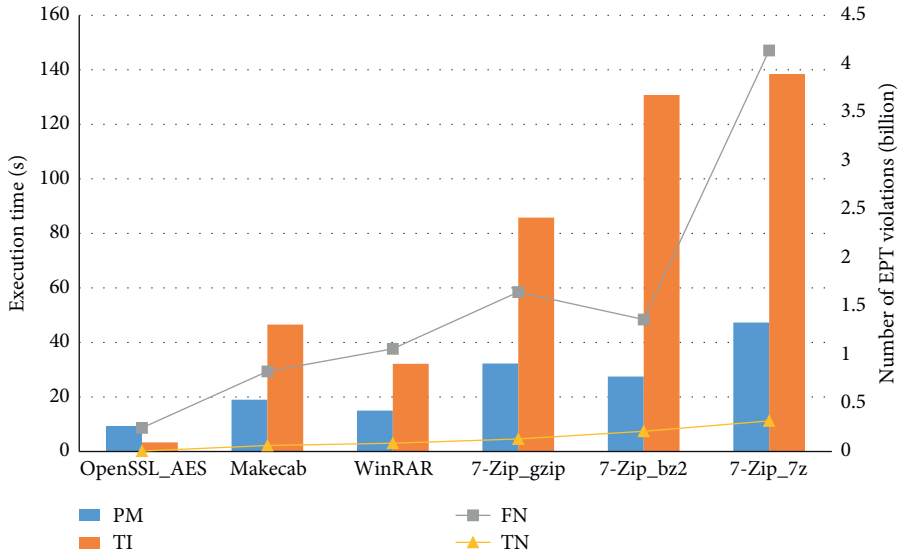


Figure 11: Relations between performance and EPT violation caused by logging in the third specific analysis approach.

5.2. Case Studies

5.2.1. Effects against the Antianalysis Tricks. Because no components are exposed in user mode, common antianalysis tricks would be difficult to stop the analysis framework, for example, detecting the process names and the existence of debuggers. Then the *al-khaser* application was used as the analysis target [1], which is a demonstration program with abundant antianalysis tricks used by malware, such as antisandbox and anti-VM. When the test program runs, the result of each detection term will be outputted as "*GOOD,*" if it does not find the corresponding analysis feature. Through several experiments, all the results show that only the term "*if CPU hypervisor field is set using cupid(0x1)*" is "*BAD.*" It means the processors are currently running in the virtualization environment, and the result is obtained by the execution of *cupid* instruction. Currently we do not sedulously intercept its execution and conceal the real result. For the known antianalysis tricks, we can also take the countermeasures for protection in the privileged mode.

5.2.2. Analysis of the Real Malicious Code. We utilized the framework to analyze the real malware sample called *Trickbot*. When the sample starts to run, it reads the client identification from the precreated file *client_id* on the infected host, which is subsequently sent to the remote server through the https protocol. We try to analyze this process and then mark the content reading from *client_id* as the taint source and detect the tainted data from the packet that will be sent out later. To accomplish the communication, we also build a local server and redirect the traffic of the sample to that, because the original communication server is invalid.

Then, based on the core parts of *libdft* [24], we further build the taint analysis engine in the kernel module of guest OS. The shadow memory of size 256 MB is allocated in kernel mode to store the taint state of user mode memory addresses of target program in guest OS. In the partial reexecution of instructions that have been executed, the lightweight instrumentation is used to record the context state before each instruction; then the dynamic taint analysis is synchronously performed. Additionally, some system services of the guest OS are hooked such as *NtReadFile* and *NtDeviceIoControlFile*. Then the experimental result shows that the tainted encrypted data can be successfully detected in the function call of *NtDeviceIoControlFile*. When the first specific analysis approach is adopted, it takes about 10 s for the program execution from reading the file to sending the packet. But the time consumption is shorten to about 2 s by the second analysis approach, and the performance can be greatly improved. The time consumption mainly concentrates on the process of establishing the SSL session, which involves large amount of encryption and decryption in a short time. It is tolerable and no exceptions occur in the subsequent running of target program.

5.2.3. Analysis between Multiple Processes. The framework can directly analyze the running program on the target OS, such as service program and system process. Moreover, the multiple processes without parent-child relationships can also be analyzed together. Finally, we give a study on the analysis of the file downloading process using the program *scp* of *OpenSSH*. In fact, *scp* first runs another program, *ssh*, to communicate with the target server and then receives the data from *ssh* through the named pipe mechanism and writes data to the local file. In this experiment, the size of the file to be downloaded is 1 kB; when the data is received from the network in the kernel function *nt!IopfCompleteRequest*, we set it as the taint source and detect the tainted data written to the local file in function *nt!NtWriteFile* successfully.

Unlike the prior analysis, we mark the memory taint based on physical memory addresses, because there are multiple processes to be analyzed at the same time. During the analysis, about 8 MB of memory is dynamically allocated

as the shadow memory, which is low in real scenarios. The whole analysis takes about 130 ms, since the number of EPT faults is small, about half a million. In addition, it should be noted that the named pipe accomplishes the data sharing between different processes through memory copies in the system kernel; we intercept the operations of reading and writing user buffers occurring in kernel function *nt!mempcy* without additional interceptions, and the taint propagation can also be correctly performed. Moreover, it is easy for us to find that the pipe finally implements the data sharing by the functions *npfs!NpAddDataQueueEntry* and *npfs!NpReadDataQueue* without deep studies and reverse engineering.

6. Discussion

This paper integrates the hardware virtualization with the system kernel, which may lose some stealthiness compared with schemes assisted by pure hardware. But it can improve the performance of automated analysis and the applicability of deployment, while lowering the difficulty in acquiring semantics information. The proposed method can take over analysis at any time when the target program is running and can directly deploy analysis in the target operating environment. It can ensure the stable execution of target program and can be targeted at large applications and programs with multiple processes at runtime, to which traditional instrumentation may not be applicable, as it may cause the target program to crash, for example, browser programs.

This paper focuses on instruction-level automated analysis, so when the number of intercepted memory operations grows, the performance of the target program will decrease. Thus, analysis of programs that are particularly sensitive to slowdown can lead to failure. To cope with such condition, optimizations can be made in accordance with specific features of target program, for example, skipping the fixed buffer used for reading and writing large files and only intercepting the specific memory areas. Moreover, some non-memory-access instructions can be intercepted combined with the executable permissions of EPT, or invisible breakpoints can also be adopted [18].

Limited by current test conditions, only horizontal comparison with dynamic instrumentation is performed, but the absolute analysis efficiency can be improved by better hardware. Although this paper proposes three specific analysis approaches, the focus is on the first two, which can perform accurate and fine-grained analysis online, while the last one involves offline analysis, which will increase complexity and uncertainty. The analysis methods in this paper are real-time and tightly coupled, but the decoupled method can also be integrated based on the interception framework to reduce the performance overhead [22].

As the Windows platform is not open-source, the functions involved in the module for managing program context may vary according to the version update of kernel; then the extra manual work should be performed to adapt the framework to different environments. Additional researches and experiments are also needed for the 64-bit platform.

7. Conclusions

This paper proposes an automation analysis method for binary programs based on hardware virtualization features and kernel assistance. This method uses hardware features to intercept the execution of the target program in real time in place of the traditional dynamic binary instrumentation method. In the analysis stage, three specific analysis approaches are proposed accordingly to analyze all the instructions that have been executed, which can perform data taint analysis at the instruction level.

Experiments in real environment show that the proposed methods and flexible designs can reduce the impact on the performance of target program and improve user experience. They can be used for automatic analysis of programs in actual environment with practical advantages under analysis scenarios such as graphically intensive programs, multiprocessed programs, and running programs. In the near future, we will focus on the performance optimizations and conduct experiments on the new platform.

Data Availability

No data were used to support this study.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this article.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (General Program) under Grant no. 61572253 and the Innovation Program for Graduate Students of Jiangsu Province, China (Grant no. KYLX16_0384).

References

[1] Al-Khaser, 2018, https://github.com/LordNoteworthy/al-khaser.

[2] Analysis of Anti-Analysis, 2018, https://github.com/yellowbyte/analysis-of-anti-analysis.

[3] J. Pan and Y. Zhuang, "PMCAP: a threat model of process memory data on the windows operating system," *Security and Communication Networks*, vol. 2017, Article ID 4621587, 15 pages, 2017.

[4] I. Korkin and S. Tanda, "Detect kernel-mode rootkits via real time logging & controlling memory access," in *Proceedings of the Conference on Digital Forensics, Security and Law (ADFSL)*, Daytona Beach, FL, USA, May 2017.

[5] N. Nethercote and J. Seward, "Valgrind," *Electronic Notes in Theoretical Computer Science*, vol. 89, no. 2, pp. 44–66, 2003.

[6] G. T. Sullivan, D. L. Bruening, I. Baron, T. Garnett, and S. Amarasinghe, "Dynamic native optimization of interpreters," in *Proceedings of the ACM SIGPLAN 2003 Workshop on Interpreters, Virtual Machines and Emulators*, pp. 50–57, San Diego, CA, USA, June 2003.

[7] C.-K. Luk, R. Cohn, R. Muth et al., "Pin," *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 190–200, 2005.

[8] Frida, 2018, https://github.com/frida/Frida.

[9] J. Kirsch and Z. Zhechev, "Pwning Intel pin reconsidering Intel pin in context of security," in *Reconsidering Intel Pin in Context of Security 2018*, Montreal, Canada, June 2018.

[10] M. Polino, A. Continella, S. Mariani et al., "Measuring and defeating anti-instrumentation-equipped malware," in *Proceedings of the 14th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pp. 73–96, Bonn, Germany, July 2017.

[11] J. Zeng, Y. Fu, and Z. Lin, "PEMU: a pin highly compatible out-of-VM dynamic binary instrumentation framework," in *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, pp. 147–160, New York, NY, USA, March 2015.

[12] A. Henderson, L. K. Yan, X. Hu, A. Prakash, H. Yin, and S. McCamant, "DECAF: a platform-neutral whole-system dynamic binary analysis platform," *IEEE Transactions on Software Engineering*, vol. 43, no. 2, pp. 164–184, 2017.

[13] J. Pan, G. Yan, and X. Fan, "Digtool: a virtualization-based framework for detecting kernel vulnerabilities," in *Proceedings of the 26th USENIX Security Symposium (USENIX Security'17)*, pp. 149–165, Vancouver, Canada, August 2017.

[14] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: malware analysis via hardware virtualization extensions," in *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS'08)*, pp. 51–62, Alexandria, VA, USA, October 2008.

[15] E. Bauman, G. Ayoade, and Z. Lin, "A survey on hypervisor-based monitoring," *ACM Computing Surveys*, vol. 48, no. 1, pp. 1–33, 2015.

[16] P. Ferrie, Attacks on Virtual Machine Emulators, 2018, https://www.symantec.com/content/dam/symantec/docs/security-center/white-papers/attacks-on-virtual-machine-emulators-07-en.pdf.

[17] F. Zhang, K. Leach, A. Stavrou, H. Wang, and K. Sun, "Using hardware features for increased debugging transparency," in *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 55–69, San Jose, CA, USA, May 2015.

[18] Z. Deng, X. Zhang, and D. Xu, "SPIDER: stealthy binary program instrumentation and debugging via hardware virtualization," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC'13)*, pp. 289–298, New Orleans, LA, USA, December 2013.

[19] Intel 64 and IA-32 Architectures Software Developer Manuals, 2018, https://software.intel.com/en-us/articles/intel-sdm.

[20] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, February 2005.

[21] K. Jee, G. Portokalidis, V. P. Kemerlis et al., "A general approach for efficiently accelerating software-based dynamic data flow tracking on commodity hardware," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, February 2012.

[22] J. Ming, D. Wu, G. Xiao, J. Wang, and P. Liu, "TaintPipe: pipelined symbolic taint analysis," in *Proceedings of the the 24th USENIX Security Symposium (USENIX Security'15)*, pp. 65–80, Washington, DC, USA, August 2015.

[23] D. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall, "TaintEraser," *ACM SIGOPS Operating Systems Review*, vol. 45, no. 1, pp. 142–154, 2011.

[24] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, "Libdft," *ACM SIGPLAN Notices*, vol. 47, no. 7, pp. 121–132, 2012.

[25] K. Jee, V. P. Kemerlis, A. D. Keromytis, and G. Portokalidis, "ShadowReplica: efficient parallelization of dynamic data flow tracking," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS'13)*, pp. 235–246, Berlin, Germany, November 2013.

[26] J. Ming, D. Wu, J. Wang, G. Xiao, and P. Liu, "StraightTaint: decoupled offline symbolic taint analysis," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 308–319, Singapore, September 2016.

[27] X. Wang, R. Ma, B. Dou, Z. Jian, and H. Chen, "OFFDTAN: a new approach of offline dynamic taint analysis for binaries," *Security and Communication Networks*, vol. 2018, Article ID 7693861, 13 pages, 2018.

[28] S. Banerjee, D. Devecsery, P. M. Chen, and S. Narayanasamy, "Iodine: fast dynamic taint tracking using rollback-free optimistic hybrid analysis," in *Proceedings of the 2019 IEEE Symposium on Security and Privacy*, pp. 490–504, San Francisco, CA, USA, May 2019.

[29] A. Davanian, Z. Qi, Y. Qu et al., "DECAF++: elastic whole-system dynamic taint analysis," in *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses*, pp. 31–45, Beijing, China, September 2019.

[30] T. K. Lengyel, S. Maresca, B. D. Payne et al., "Scalability, fidelity and stealth in the DRAKVUF dynamic malware analysis system," in *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC 2014)*, pp. 386–395, New Orleans, LA, USA, December 2014.

[31] Y. Ji, S. Lee, M. Fazzini et al., "Enabling refinable cross-host attack investigation with efficient data flow tagging and tracking," in *Proceedings of the 27th USENIX Security Symposium*, pp. 1705–1722, Santa Clara, CA, USA, August, 2018.

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

[33] P. Barham, B. Dragovic, K. Fraser et al., "Xen and the art of virtualization," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 164–177, 2003.

[34] Z. Liu, J. GroBschadl, Z. Hu, K. Jarvinen, H. Wang, and I. Verbauwhede, "Elliptic curve cryptography with efficiently computable endomorphisms and its hardware implementations for the internet of things," *IEEE Transactions on Computers*, vol. 66, no. 5, pp. 773–785, 2017.

[35] C.-H. Lin, H.-K. Pao, and J.-W. Liao, "Efficient dynamic malware analysis using virtual time control mechanics," *Computers & Security*, vol. 73, pp. 359–373, 2018.

[36] C. Willems, R. Hund, A. Fobian, D. Felsch, T. Holz, and A. Vasudevan, "Down to the bare metal: using processor features for binary analysis," in *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC'12)*, pp. 189–198, Orlando, FL, USA, December 2012.

[37] S. Vogl and C. Eckert, "Using hardware performance events for instruction-level monitoring on the x86 architecture," in *Proceedings of the 2012 European Workshop on System Security (EuroSec'12)*, Bern, Switzerland, April 2012.

[38] M. B. Bahador, M. Abadi, and A. Tajoddin, "HLMD: a signature-based approach to hardware-level behavioral malware detection and classification," *The Journal of Supercomputing*, vol. 75, no. 8, pp. 5551–5582, 2019.

[39] D. Zhan, L. Ye, B. Fang, H. Zhang, and X. Du, "Checking virtual machine kernel control-flow integrity using a page-

level dynamic tracing approach," *Soft Computing*, vol. 22, no. 23, pp. 7977–7987, 2018.

[40] Z. Ning and F. Zhang, "Hardware-assisted transparent tracing and debugging on ARM," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 6, pp. 1595–1609, 2019.

[41] Intel Virtualization Technology, 2018, https://www.intel.com/content/www/us/en/virtualization/virtualization-technology/intel-virtualization-technology.html.

[42] AMD Virtualization Technology, 2018, https://www.amd.com/en/technologies/virtualization-solutions.

[43] Y. Liu, P. Shi, X. Wang, H. Chen, B. Zang, and H. Guan, "Transparent and efficient CFI enforcement with Intel processor trace," in *Proceedings of the 23rd IEEE Symposium on High Performance Computer Architecture (HPCA)*, pp. 529–540, Austin, TX, USA, February 2017.

[44] B. Cui, F. Wang, T. Guo, and G. Dong, "A practical off-line taint analysis framework and its application in reverse engineering of file format," *Computers & Security*, vol. 51, pp. 1–15, 2015.

[45] HyperPlatform, 2018, https://github.com/tandasat/HyperPlatform.

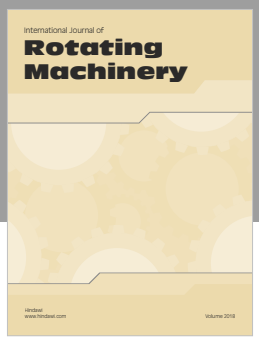
[46] Libudis86, 2018, http://udis86.sourceforge.net/manual/libudis86.html.

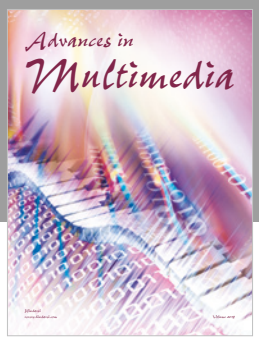
[47] FFmpeg, 2018, https://ffmpeg.zeranoe.com/builds/win32/static/ffmpeg-4.0.2-win32-static.zip.

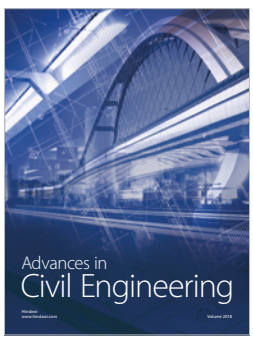
[48] Curl, 2018, https://bintray.com/artifact/download/vszakats/generic/curl-7.61.0-win32-mingw.zip.

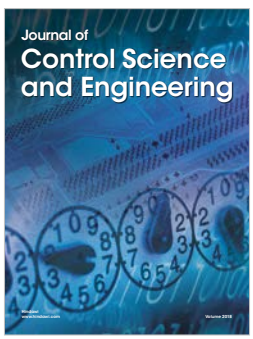
[49] 7-Zip 18.05, 2018, https://www.7-zip.org/a/7z1805.exe.

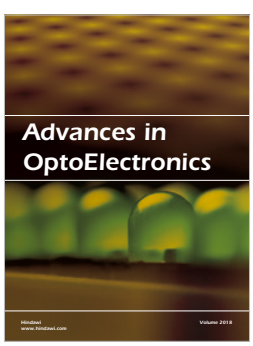
[50] Openssl 1.0.2n, 2018, https://www.openssl.org/source/openssl-1.0.2n.tar.gz.

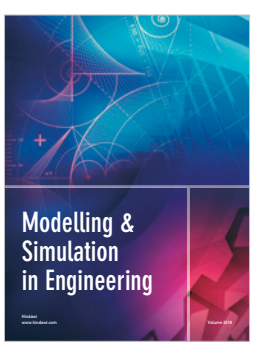


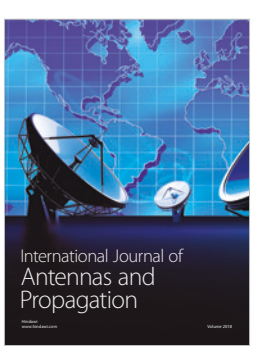


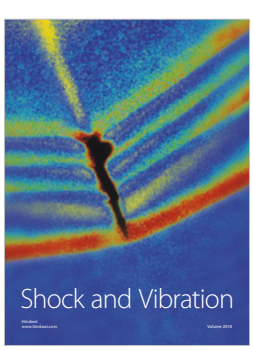


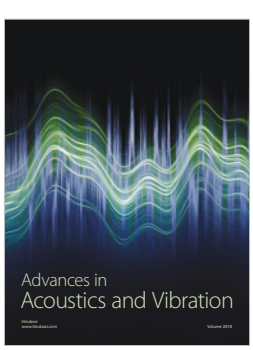












Hindawi

Submit your manuscripts at
www.hindawi.com