Received August 14, 2020, accepted August 29, 2020, date of publication September 3, 2020, date of current version September 21, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3021463

Lightweight and Efficient Hypervisor-Based Dynamic Binary Instrumentation and Analysis Method

JIAYE PAN[1], **ZHUANG YI**[1], **ZHAO XUE-JIAN**[2], **AND BINGLIN SUN**[1]
[1]College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 211106, China
[2]Key Laboratory of Broadband Wireless Communication and Sensor Network Technology, Ministry of Education, Nanjing University of Posts and Telecommunications, Nanjing 210003, China

Corresponding authors: Zhuang Yi (zy16@nuaa.edu.cn) and Zhao Xue-Jian (zhaoxj@njupt.edu.cn)

This work was supported in part by the National Science Foundation of China under Grant 61572253.

ABSTRACT At present, various vulnerabilities and malicious programs are still constantly threatening the system security, and in-depth analysis of legitimate applications and malicious code is an important link of security defense under the current security situation. Dynamic binary analysis is the important method in the field of binary analysis, and after years of research and development, many valuable results have been achieved, and some mature tools generated have been widely used in practice. But it still faces multiple challenges in terms of analysis efficiency, deployment, and the impact on the target program. Based on the existing research, this article proposes a new lightweight hypervisor-based dynamic binary instrumentation method, which uses the virtualization features of new processors to perform transparent and efficient execution interception of the target program, so that it can instrument the target program and redirect the original execution. To take full advantage of the interception solution, we further present a compact inline dynamic taint analysis method that enables fine-grained data flow analysis of the target program with multiple processes and kernel modules. Experiments in the real environment show that the proposed method is effective and efficient in the binary instrumentation and analysis, it can achieve a good balance in analysis performance, availability and stealthiness, and can be applied to the practical analysis scenarios.

INDEX TERMS Binary program, dynamic analysis, hypervisor, instrumentation, taint analysis.

I. INTRODUCTION

Today, the cyber security situation is more complicated, various kinds of advanced threats are constantly emerging, and endangering the security of the target information system, especially the advanced persistent threat. It is important to analyze the vulnerabilities and malicious code that are related to threat activities in depth, in order to extract the threat intelligence information. On the one hand, many researchers focus on the security analysis of the legitimate software, to find out the existing vulnerability and submit it to the manufacturers for repair in advance. On the other hand, researchers face a large demand for malicious code analysis to prevent further malware spread and infection. In most cases, security analysts face the analysis of binary programs, where static analysis and dynamic analysis are commonly used methods [1]–[3]. To improve the analysis efficiency, stealthiness and applicability is an issue that has been continuously researched with the development of hardware and software technology [4]–[8].

Static analysis usually can obtain the higher code coverage and better performance, but there are difficulties in analyzing the dynamic generated code [4]. Meanwhile the dynamic analysis method is widely used in practical analysis due to the ability to detect the actual execution path and behaviors of the target program [9], [10]. In this case, the coarse-grained dynamic analysis often tracks the process, file and network operations of the target program based on the function level [11]. The fine-grained method can be used to analyze the target program at the instruction level, mostly based on the dynamic binary instrumentation (DBI). There have been a lot of research in the past many years, which produces some practical results, such as Pin [12], DynamoRIO [13], Valgrind [14]. In recent years, new tools such as Frida [15]

The associate editor coordinating the review of this manuscript and approving it for publication was Dongxiao Yu.

and QBDI [16] can support multiple platforms and analysis code writing by scripting languages such as JavaScript and Python, which further simplifies the development of analytical tools. Because dynamic instrumentation methods can be used to analyze the target program at the instruction level, it firstly needs to intercept the program execution and rewrite the native code dynamically, mainly including two levels. One is analyzing the program running in user mode inside the operating system (OS), based on the method similar to the just-in-time (JIT) compilation, which translates the native instructions and builds the equivalent executable code in the new memory space. At this point, the instrumentation tool often exists as the parent process of the target process, and inject some related functional modules into the target process space [6]. The runtime performance of some programs may be significantly reduced, due to the large amount of execution interception and instruction translation in the instrumentation process [17], [35]. The other is building the analysis framework outside the operating system, which is usually based on the emulator or virtualization methods such as QEMU, VMI to implement the translation and analysis of the target instructions [5], [29], [32]. Although some frameworks can support fine-grained analysis in the system-wide, it is difficult to obtain abundant semantic information of the target program at runtime, and often there also exists problems such as analysis performance and deployment.

Actually, dynamic binary instrumentation is widely used in malicious code analysis, program security analysis, software protection and other fields [18]–[22], and lots of ideas and methods for program analysis are implemented and verified based on it [33]–[37]. Although it faces some problems and challenges in practical applications, such as: affecting the runtime performance of the program, occupying large amount of memory resources, and confronting the anti-analysis techniques [4], [23], it is still worth us to take countermeasures to make it effective in the actual analysis scenario [4], [7].

Therefore, based on the existing research work, we proposes an efficient dynamic binary instrumentation and analysis method named HyperAnalyzer (HA), which intercepts and instruments the target program on the basis of the lightweight virtualization framework, extended page table (EPT) and virtualization exception (#VE) mechanisms of the new processor, so that the fine-grained dynamic analysis at the instruction level can be performed. Compared with existing analysis methods, the proposed method can intercept the program execution and accomplish the code instrumentation more efficiently, which is easy to deploy and apply in the target operating system, and achieves a better balance in the aspects of performance overhead, transparency, deployment, and the impact on the running program. It intercepts the execution of target program and constructs the corresponding analysis code mainly in kernel space, which can easily obtain rich semantic information at runtime, and will be more advantageous in countering the anti-analysis techniques than in user mode. In addition, the analysis framework can be directly deployed in the target system, and can support the analysis of both the user-mode code and kernel modules.

The main contributions of this article are as follows:

Firstly, this article proposes the new lightweight dynamic binary instrumentation and analysis method, which can effectively intercept and instrument the target program at runtime based on the latest hardware virtualization technology, then further implements the specific fine-grained analysis. It can instrument the entire or part of the program from kernel space, obtain the semantic information easily and reduce the impact on the runtime environment of target program.

Secondly, on this basis, this article further presents a compact inline analysis method for the dynamic taint analysis, which is closely combined with the proposed interception solution. It can be used to perform the dynamic taint analysis of the target program more efficiently, so that we can accomplish the complicated analysis more easily including the data flow propagation between multiple processes, the interaction between user mode programs and kernel modules.

Finally, this article implements the prototype framework of the proposed method on the Windows platform, and uses multiple kinds of real programs to evaluate the performance of instrumentation and specific analysis. We also verify the effectiveness of dynamic taint analysis and kernel driver analysis by using the actual scenarios.

II. METHOD DESCRIPTION AND FRAMEWORK DESIGN

Similar to the traditional method, The proposed method first needs to intercept the execution of the target program, but the difference is that here we accomplish the interception of code execution by controlling the access permissions of corresponding code pages of the target program based on the EPT mechanism [26], [31]. In fact, the processor with EPT mechanism has been widely available in commercial products, and many studies have applied its features to program debugging analysis, software protection, and so on [26], [30], [32]. The architecture overview of the proposed method is shown as Fig. 1, it first deploys and loads a specific kernel module in the target operating system, and enables the virtualization function supported by the target processors, then converts the operating system to run as the virtual machine (VM) or guest OS mode. Therefore, the processor model of the target machine needs to have the relevant features that support the successful application of the proposed method, which can be generally met on the prevailing processors and personal computers.

The modules and workflows involved in the method are mainly located in kernel space of the target system, as shown in Fig. 1. There are multiple monitors in the kernel that track the behaviors of target program, and the monitoring mainly depends on the hooking of the relevant system calls and internal kernel functions in the guest OS. The process thread monitor can capture the creation and exit events of the processes and threads associated with the target program in real time, and correspond it with other relevant runtime information, also manage the program execution

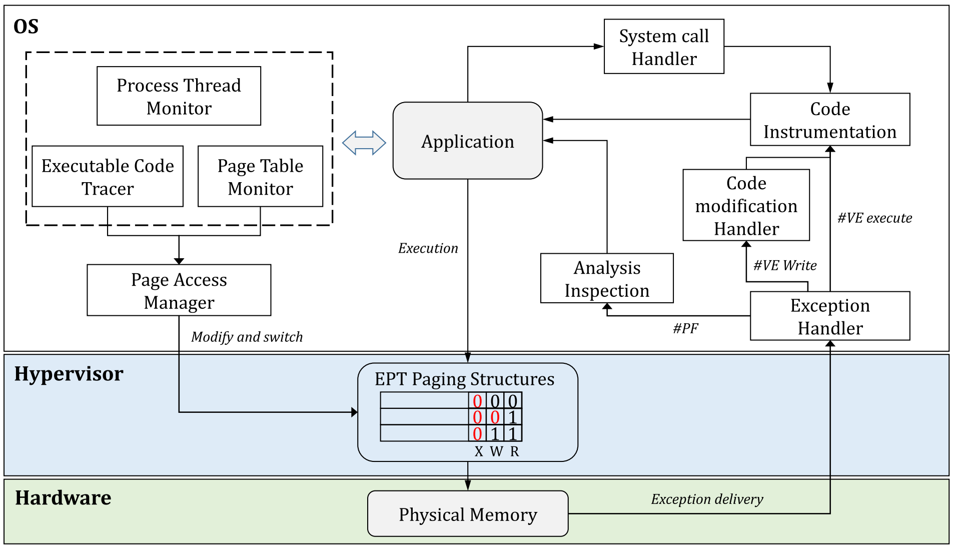


FIGURE 1. Architecture overview of HyperAnalyzer.

switch between native mode and analysis mode at runtime. The module in the hypervisor is mainly responsible for supporting the management and configuration of the running state of virtualization environment, and is also used to handle the non-maskable or special VM exit events. There are two main tasks included in the analysis, one is that when the program generates the virtualization exception or fault during the execution, we parse and instrument the relevant code in the virtualization exception handler. The instructions related to the system calls should also be intercepted and handled during the program execution. The second is the continuous tracking of the executable code loaded by the target program at runtime, and modifying the access permission of the corresponding guest physical address in the EPT paging structures, so that all the code execution of the target program can be intercepted and redirected.

A. EXECUTABLE PAGE TRACKING

The execution of target program involves user space and kernel space, although most programs only contain modules in user mode. In this situation, the code executed in kernel mode by the target thread belongs to the operating system kernel or device drivers. To simplify the complexity and improve accuracy, we track the target code executed in user mode and kernel mode respectively. Because program code is stored in shared memory pages on most system platforms, that is, the same memory pages are simultaneously mapped into the virtual memory spaces of multiple processes or the kernel. Therefore, it is necessary to continuously track and monitor the code loading of the target program to distinguish between private code pages and shared code pages, while monitoring the changes of memory access permissions of related pages, such as read, write and execute, in order to make correct analysis of the target program, reduce the impact on the whole system and other applications, and control the resource overhead during the analysis. When the process of the target program is created, some physical pages of executable code have been already allocated in the memory space. For these pages, we can get the physical addresses by traversing directly the process page table before the main thread is executed, and then continue to track the changes of the code that is dynamically loaded at runtime of the target program.

For tracking the executable code in user mode, we can simply achieve this by hooking the function calls associated with the page allocation and access permission modification of the operating system. Specifically, the kernel functions for memory allocation are hooked to monitor the physical page allocation in user space, and the system internal functions for modifying memory access permissions can also be hooked to track the allocation of virtual memory and the changes of access permissions related to the target process. On the basis of monitoring the process activities and memory state changes of the target program, we can distinguish the memory pages between code and data, private and shared. Then, to reduce the impact on other programs and the whole system, we replicate the shared pages in the target process, and remap the original virtual addresses to the replicated physical pages in the page table, then set the access rights of the new pages to non-executable in the EPT entries. Therefore, the execution of any instruction of the target program will generate a virtualization exception, which will be further delivered to the guest OS and handled in the kernel space. As shown in Fig. 2, a new physical page is allocated for each shared page loaded by the target process and is used to replace the original mapping entry in the page table. Then the original page and the new page are kept in sync, which means that the data of the two are consistent. In addition, if the program attempts to modify the original shared page, it will also trigger the copy-on-write mechanism of the system, so we should

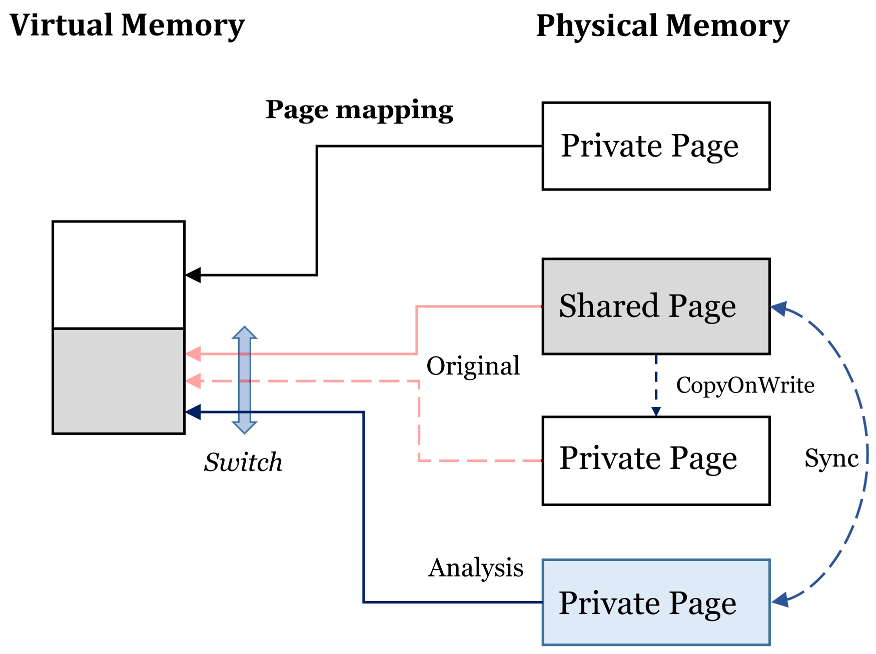


FIGURE 2. Remapping of the shared memory pages.

have a special treatment for this situation and ensure that the operating system still performs normal management of the page. Therefore, when the target program writes to the original shared page, the analysis framework can also capture the operation and restore its mapping to the original physical page in the page table, then leave the situation to the system for processing. After done, the newly allocated private page will also be included in the scope of interception. Finally, when the analysis of the code page is completed, the mapping of its physical page in the page table can be restored to its original state as needed.

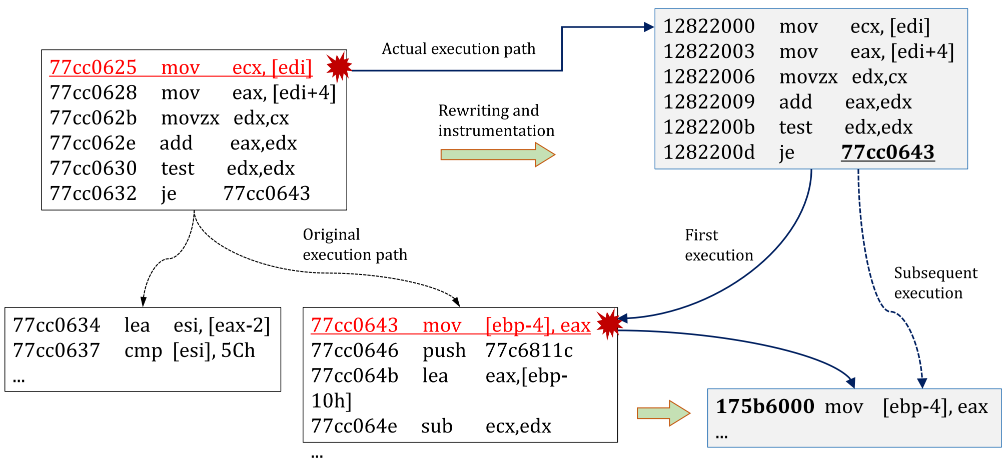
For easy lookup and tracking, we establish a direct mapping table between the base addresses of the original shared page and the newly allocated private page. The memory space of the mapping structure itself is allocated in kernel space, for 32-bit platform, 4 MB of memory space is occupied, the top 20 bits of base address of the original shared page are used as an index to find the corresponding page address and relevant information through the mapping table. It is important to note that because the shared pages are localized by modifying the process page table directly, the modification is still imperceptible to the execution of target program, but to avoid affecting the system's normal management of memory working set of the target process, we also need to modify the physical page database structure (_MMPFN) related to the original shared page in the kernel. We set the values of some fields in the corresponding structure such as the page type (*type*) and working set index (*WsIndex*), still treat it as a shared page and continue to track its state changes.

The tracking of executable code in kernel mode can be carried out directly based on the monitoring of accesses to the page table of target process. Because the memory resource and code in kernel space is also shared, and it is stable in most cases, we can directly replicate the target page and remap the original virtual address to the newly allocated physical page in the page table for the code of system kernel and loaded drivers that need to be analyzed, referring to the previous paragraph. For other dynamic loading code, it can be tracked based on the interception of write access to the page table of target process [26]. Another thing to note is that in order to ensure systemic performance and stability, this is just tracking the kernel code executed in the target thread context as needed, rather than tracking the code that is executed in kernel space of the entire system.

B. EXECUTION INTERCEPTION AND REDIRECTION

On the basis of localization and tracking of the code executed by the target program, the program execution can be intercepted based on the access control for fetching instructions from the physical pages which is supported by the EPT mechanism. When an access violation is triggered, it can be further delivered to the guest OS according to the feature of virtualization exception in the new generation of processors. So if the access permissions of target physical pages in EPT entries are modified to non-executable, any instruction execution of the target program will generate a virtualization exception.

However, frequent virtualization exceptions will also cause a sharp decline in the performance of the system and program [31], so we use the method called "redirection on exception" to avoid this situation. In the handling of an exception, the basic block of the program is used as a unit to parse and rewrite. When the execution of an instruction of the basic block is intercepted, we disassemble and parse the next instructions until a branch instruction, rebuild the equivalent code blocks in the new memory area, and then redirect the



```
77cc0625   mov     ecx, [edi]
77cc0628   mov     eax, [edi+4]
77cc062b   movzx  edx,cx
77cc062e   add     eax,edx
77cc0630   test    edx,edx
77cc0632   je      77cc0643
```

```
12822000   mov    ecx, [edi]
12822003   mov    eax, [edi+4]
12822006   movzx  edx,cx
12822009   add    eax,edx
1282200b   test   edx,edx
1282200d   je     77cc0643
```

```
77cc0634   lea   esi, [eax-2]
77cc0637   cmp  [esi], 5Ch
...
```

```
77cc0643   mov    [ebp-4], eax
77cc0646   push   77c6811c
77cc064b   lea    eax,[ebp-
10h]
77cc064e   sub    ecx,edx
...
```

```
175b6000  mov   [ebp-4], eax
...
```

FIGURE 3. Execution interception and redirection on the basic block granularity.

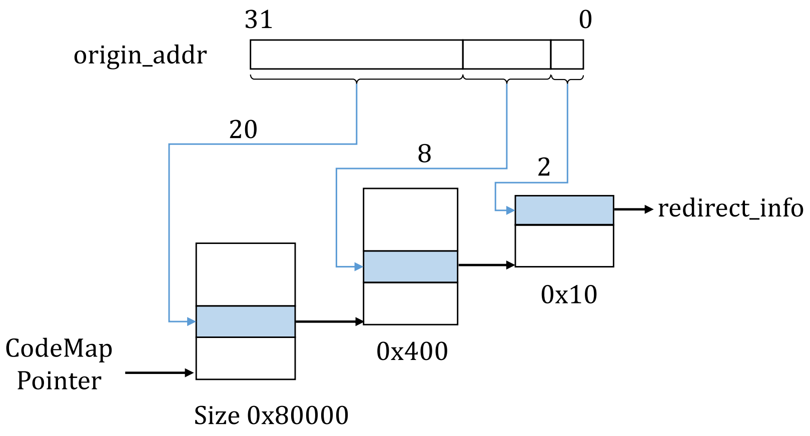


FIGURE 4. Mapping of the original address to the redirection information structure of generated code.

original execution to the newly allocated code that is called as generated code here. As shown in Fig. 3, the execution of the instruction at 77cc0625 generates an exception, then the subsequent execution is redirected to 12822000 that points a newly allocated code block, into which additional analysis code can also be inserted according to the specific analysis requirements. Then the subsequent execution generates another exception at 77cc0643, which is handled the same as mentioned above. However, it is important to note that in this exception, we will modify the direct jump address 77cc0643 of the last block to the newly allocated address 175b6000 so that the subsequent execution can directly jump from that block to the newly allocated code to avoid another exception.

The execution interception solution makes full use of the hardware performance, compared with the JIT method, it can greatly improve the transparency and efficiency of interception. In addition, this method can also capture the dynamic code modification in the writable page, usually modified by the program self at runtime. In specific applications, we can also only intercept part of the code pages, such as the specific modules or functions. Because the main components are built at the kernel and hypervisor levels, part of the kernel address space can also be used to store the generated code. The framework is still highly transparent to the analysis of the program in user mode, which can be used to online analyze multiple processes simultaneously, and can safely restore the target program from being analyzed to native execution.

1) EXECUTING CODE GENERATION

As mentioned above, the original code is parsed and instrumented when the execution of target program generates the virtualization exceptions, then the execution will be redirected to the newly generated code. Therefore, in the exception handling process, we first check whether the basic block that cause the exception has been parsed. In order to improve the search efficiency from the original instruction address to the newly generated executable code, we construct a three-level index structure, denoted as *CodeMap*, which is similar to the page table structure, as shown in Fig. 4. So if an exception is generated at the address *ins_addr*, we can obtain the corresponding redirection information structure denoted as *redirect_info* through *CodeMap[ins_addr]*, which contains the original address, the address of generated code,

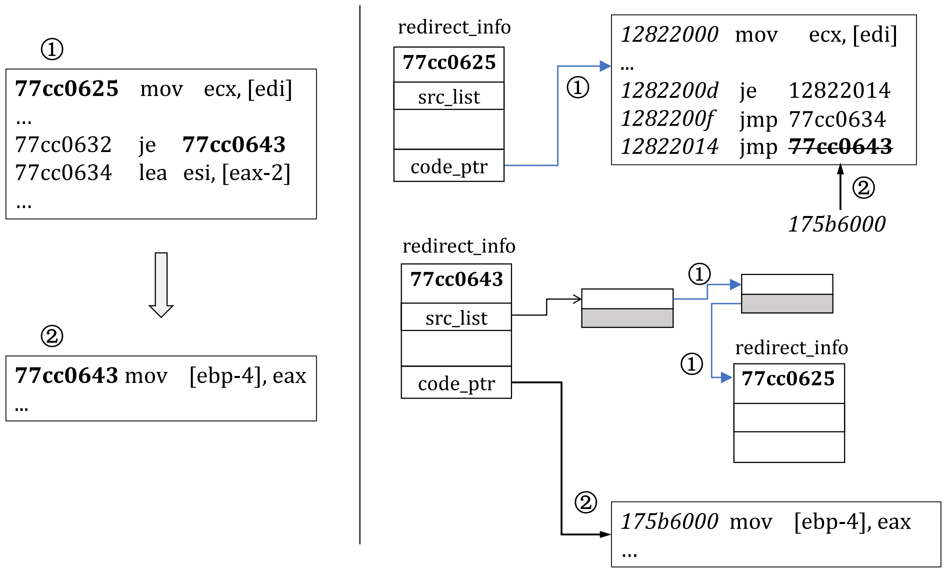


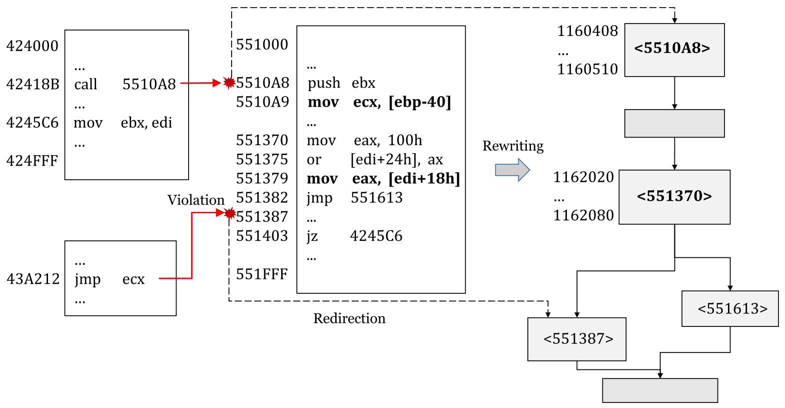
FIGURE 5. Construction of the generated code for the direct jump instruction.

and so on. For 32-bit platform, the primary index table needs to occupy 4 MB of additional memory space, and the space consumptions of the latter two levels structures depend on the size of the code actually loaded by the target program at runtime.

In the handling of virtualization exception, if the corresponding basic block has been parsed, then the execution of the target thread is directly redirected to the newly generated executable code. For the unparsed basic block, first we disassemble and parse the binary code, then allocate memory areas in the target process space to store the newly generated code, and also build the corresponding redirection information structure and create the mapping entry. Finally, the branch instruction at the end of the basic block is specially handled, including two cases, direct jump and indirect jump.

In the case of direct jump, the destination address is derived from the original code and the relative offset of the jump target will change when the execution of original code is redirected, so we need to re-establish the link between different generated executable code blocks to reduce the number of virtualization exceptions. In the generated code, the conditional jump instruction is reconstructed with the unconditional jump instructions, which will link different generated code blocks to ensure that they are continuously executed, and allow the processor to perform the branch prediction when executing the code, and improve the execution speed possibly. As shown in Fig. 5, the left side is the sequential execution of two program basic blocks, which will generate two exceptions; the right-hand side is the partial procedure of exception handling. When the exception is handled at the first time, the generated code is allocated and built, and the original jump instruction is also transformed. In addition, the corresponding structure *redirect_info* is pre-allocated for the basic blocks that maybe executed subsequently, and the *redirect_info* of current basic block is also added to the *src_list* field in the pre-allocated structures. In this way, when the instruction of subsequent basic block is executed and intercepted, we can get all the other basic blocks that may be executed to the basic block generating the current exception through its corresponding *src_list* field, and update unconditional jump addresses of the corresponding generated code. As shown in Fig. 5, in the first exception handling, the generated code for the original block starting at 77cc0625 is created, and the corresponding *redirect_info* of 77cc0625 is added to the *src_list* of the *redirect_info* structure corresponding to 77cc0643. Then, in the second exception handling, apart from creating the generated code for the original block starting at 77cc0643, the jump target address 77cc0643 of the generated code corresponding to the original block starting at 77cc0625 is modified to 175b6000 so that the subsequent execution will directly reach the desired code and avoid the additional exceptions. In fact, there is also a more direct way to obtain the basic block executed last by the target thread, which is to insert the additional instruction in the newly generated code to record the execution information, and it slightly increases the size and execution overhead of the target code.

Indirect jump instructions such as *ret*, *call ecx* also need to be transformed. First, the jump target address is obtained dynamically at runtime, then the corresponding generated code can be found from the *CodeMap* structure and executed as the new jump target later, which can be implemented through a small piece of assembly code, so the number of virtualization exceptions can be further reduced. Because the basic block including the jump target address may not yet be executed, in this situation, the address of generated code stored in *redirect_info* is still the original address of the program's basic block, and it will be updated in the handling process of next virtualization exception. For the instruction such as *sysenter*, *int*, which will lead to the privilege level



```
424000    ...
42418B    call   5510A8
          ...
4245C6    mov    ebx, edi
          ...
424FFF

43A212    ...
          jmp    ecx
          ...
```

```
551000    ...
5510A8    push   ebx
5510A9    mov    ecx, [ebp-40]
          ...
551370    mov    eax, 100h
551375    or     [edi+24h], ax
551379    mov    eax, [edi+18h]
551382    jmp    551613
551387    ...
551403    jz     4245C6
          ...
551FFF
```

FIGURE 6. Diagram of interception and analysis for part of executable pages.

switching of the processor, it can be also transformed the same as the above case.

2) SYSTEM CALL HANDLING

Because there exists the user callback and asynchronous procedure call (APC) mechanism in the operating system, when the user thread returns from the system kernel to the user space, it may not sequentially execute the instruction that follows the system call instruction leading to the switch of privilege levels before. For example, when the thread returns from the kernel on the Windows platform, it first enters into the function *ntdll!KiUserApcDispatcher* if a user APC call is present. Since we have tracked all the executable code of the target program, the above execution can also be intercepted, but it will also generate a non-executable virtualization exception first when the target thread returns from the kernel. To further reduce the number of exceptions, additional processing can be done on the return procedure of system calls. On the Windows system, for the functions such as *nt!KiServiceExit* that involve the returning from kernel to user space, the interceptor can be placed before the *iretd* or *sysexit* instruction in these functions. The handling process is similar to the last section, which first obtains the runtime return address, and then finds the corresponding address to be executed actually through the *CodeMap* structure.

3) EXECUTION MODE SWITCHING

If the entire target program or most of the modules are analyzed, the executable code pages are tracked when the target program starts to run, then all the instructions executed can be parsed and rewritten. If only a small amount of code needs to be instrumented and analyzed, we can forbid the execute permissions of physical pages that contain the target code only in the EPT entries, and then instrument all the executed code which is involved in these pages.

As shown in Fig. 6, we intercept and analyze the execution of the code located in the page with the base address 551000 after handling lots of non-execute virtualization exceptions, the original code page will be transformed to the generated code blocks in the right-hand of the figure, in which the specific analysis code of the target instructions can also be placed and executed without generating any virtualization exception. However, the native execution to the address 551000 from the code outside the page will still generate an exception, but the impact of analysis on the running program is actually reduced, since the program mostly executes the native code. Moreover, if the program execution still generates a large amount of exceptions, it may also affect the runtime performance of the target program and the operating system, as shown in Fig. 6, if the instruction at 43a212 is executed frequently. In this case, the fault profiler can be added to the exception handling procedure to count the number of exceptions occurred at a specific address, and when it exceeds a threshold, we continue to intercept and rewritten the source page from which the execution jumps to the address, in order to reduce the performance impact of excessive exceptions. If we only need to analyze a few instruction addresses, as shown in Fig. 6, the instructions are located at 5510a9 and 551379, similar to the above, we then intercept and instrument the entire page that contains these instructions, but only instrument a few basic blocks of the program and insert the specific analysis code, while other original code of the page remains unchanged in the newly generated code blocks and the branch instructions also need to be handled.

When the target program is prepared to be analyzed, but it has already been running normally, then the instruction that generates the first non-executable exception may not be located at the beginning of the program's basic block, which is likely to be in the middle. In this case, we still take the fault address as the starting point of a basic block, then parse and instrument the code forward until the branch instruction.

In the analysis process, if we want to exit the analysis for some code pages, then configure the corresponding pages to be executable in the EPT entries. When the execution of the target thread jumps out of the generated code and enters into the original code again, the program continues to run natively.

4) SELF-MODIFYING CODE

Some special programs will modify the code that has been executed at runtime, usually presented as self-modifying code, which is more common in advanced malware or JIT

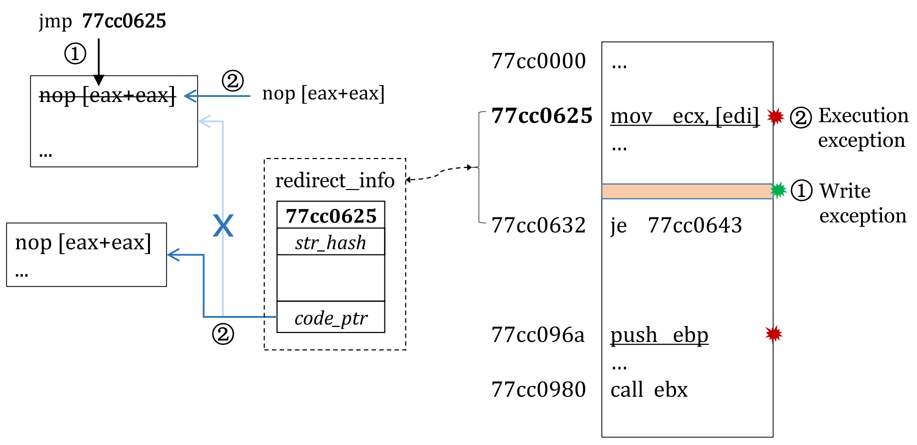


FIGURE 7. Interception and analysis of the dynamically modified code. When the non-writable exception occurs, the generated code that has been built in the page is rolled back, as shown as symbol ①; The non-executable exception will be generated again when the execution above continues, as shown as symbol ②, in the exception handling, check if the basic block is changed, and the corresponding generated code will be rebuilt if it is modified, otherwise, restore the jump stub at the beginning to dummy instructions.

code optimization [4]. The analysis method should be able to track and handle this situation to adapt to the binary analysis under different scenarios. In order to achieve this goal, on the basis of the execution interception of the instructions contained in the executable page, if the page is also writable at the operating system level, we will further intercept the write accesses to the page based on EPT mechanism. As a result, with the help of the above scheme, we can timely find out how the executing code is modified at runtime, and check whether the generated code has been changed actually, then the modified original code is reparsed to rebuild the generated code. So we introduce the additional *str_hash* field in *redirect_info*, as shown in Fig. 7, to quickly determine whether the original basic block of the target program has been changed through the simple string hash algorithm.

Specifically, when intercepting the write operations to the executable page, first we obtain all the basic block information involved in this page based on the *redirect_info* structures, and then change the dummy instructions at the beginning of the generated code to a jump stub which can make the subsequent execution to the original code for each related basic block, so that subsequent execution of the basic block will generate the non-execute exception again. After that, we continue to change the executable page to be writable on the EPT level again, which will enable the subsequent writes to succeed.

When the code execution of modified page is intercepted again later, we first check whether it is located in a basic block that has been executed, and verify whether the code has been changed actually according to the string hash, then reparse and re-instrument the modified code. Finally we restore the interception of write operations on the page to continuously capture the subsequent code modification during the execution of the target program. The diagram of code modification and execution interception is shown in Fig. 7.

C. COMPACT INLINE ANALYSIS

Based on the method described above, then we can instrument and analyze the target program according to the actual requirements. In order to make full use of the interception solution proposed in this article, we continue to present the compact inline analysis method to perform the dynamic taint analysis at instruction level. In actual analysis, byte-level taint analysis can help us complete most of the fine-grained analysis tasks, and the taint sources and sinks are mostly located at the entry points of library functions or system calls. In this case, we can use more compact assembly instructions to implement the required analysis functions to avoid additional performance overhead at the higher level analysis. The analysis code is mixed with the original code, which is similar to the analysis on the traditional instrumentation, but the difference is that the analysis code here is embedded in the original code, which means that the original instructions keep the same in the final generated code.

On 32-bit Windows platform, the segment register *FS* always points to the thread environment block (TEB) structure during the thread execution in user mode, which is thread related. We can use the memory space of *TEB* that is not used to store the analysis state of general registers for each thread in the taint analysis, and save the partial context information, actually, a larger storage space can be obtained by extending the upper limit of the segment. In kernel mode, the *FS* segment register points to the processor related structure *KPCR*, through which the thread related structure *ETHREAD* can be further referenced, but the context information during the thread execution needs to be saved using the thread stack in kernel space. Because the byte-level taint analysis is used here, for the analysis state of the process memory space of target program, the same size of additional memory space is used to store the state based on the virtual address. That is, when the accessible memory page is allocated for the target program, meanwhile we allocate the corresponding memory

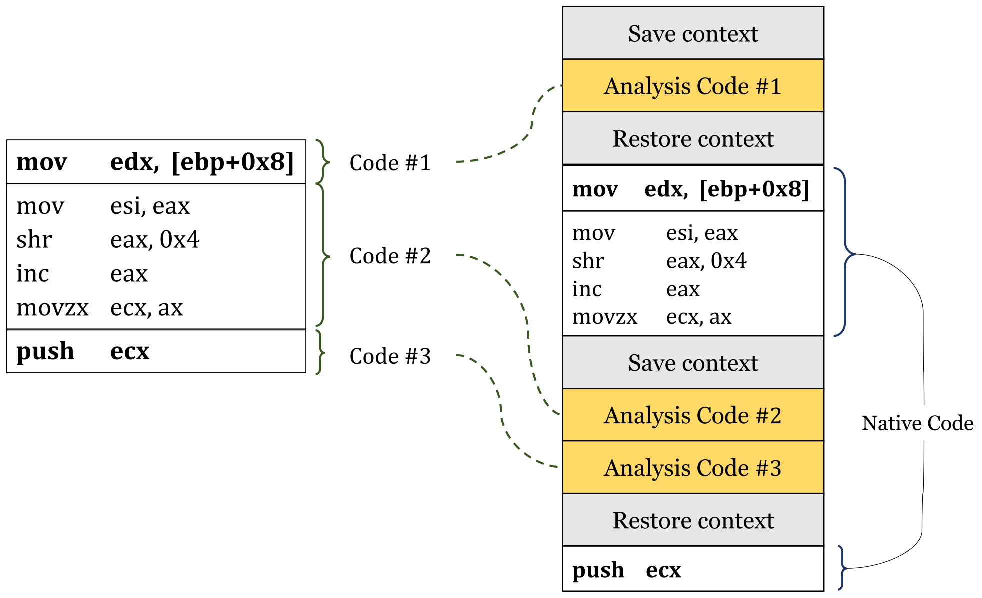


FIGURE 8. **Generation and execution of the specific analysis code.**

page in the target process space to store the analysis state and build the map entry in the mapping table, similar to the mapping structure for the generated code blocks. The entire mapping table on 32-bit platform occupies 4 MB of memory space, denoted as *StateMap* here. For the memory address *addr*, its corresponding analysis state is stored at "$st\_addr = StateMap[addr \gg 12] + addr \ \& \ 0xFFF$" in most cases. Therefore, the update process of the memory analysis state during the analysis will be simplified, and the memory space required for storing the analysis state is related to the size of memory allocated for the target program at runtime, and is more relevant to the number of physical pages actually allocated, which is relatively limited in practical applications.

In particular, when a single instruction reads or writes across pages in the execution of the target program, and if the virtual addresses allocated for storing the analysis state is discontinuous, then the analysis for the instruction execution above could lead to the reading and writing errors of the analysis state. For this situation, a guard page can be allocated following the state storage page, and then the analysis error caused by the cross-page modification can be detected and corrected. Also note that if the memory page for storing the analysis state is not yet allocated in the analysis, and a page access exception will be generated when the analysis code accesses the analysis state. For this case, we will allocate and build the state storage page in the page fault handler. The above situation often occurs when we fail to track all the memory allocation of the target program or the physical page of the virtual memory has not been allocated.

In the process of execution interception and code instrumentation, the specific analysis code for dynamic taint analysis is generated in sync, and the relevant context state of the target thread is saved before and after the execution of the analysis code, in fact, only several register values need to be saved. We can also use the thread related memory area that the *FS* segment register points to as a temporary buffer to store the register value (e.g., "*mov fs:[70h], eax*"). In order to reduce the number of context switches due to the information saving during the analysis of the target program, we appropriately delay the analysis of the code between two consecutive memory access instructions, and then place the analysis before analyzing the latter memory access instruction. The temporarily unanalyzed code mostly involves register operations, and we don't need to obtain the runtime value during the common taint analysis, but the general registers still need to be used to complete the taint propagation. Therefore, the delayed analysis can avoid saving the thread context again. As shown in Fig. 8, the analysis of the code #2 is delayed until the execution of analysis code #3, and the original thread context only needs to be saved and restored once. As discussed above, the sinks are mostly selected at the function level in the taint analysis, so the delayed analysis does not affect the timely acquisition of the analysis results, and instructions or addresses for special attention can be addressed separately.

III. PROTOTYPE IMPLEMENTATION

We implement a prototype of the proposed framework on the 32-bit Windows platform, the entire code is included in a kernel driver module, where the hypervisor component is implemented based on HyperPlatform [42], and the udis86 [43] tool is used to disassemble and parse the binary code.

For the hypervisor component, we make a small amount of changes on HyperPlatform. Specifically, in the *VmpInitializeVm* function, only one EPT structure is constructed, which is then shared by all processors. In the *VmpSetupVmcs* function, we turn off the unnecessary VM exit options (e.g. *cr3_load_exiting*), and configure the options such as *ept_violation_ve*, *enable_vm_functions* to enable the virtualization exception function. In addition, we need to set the parameters (e.g. *kEptpIndex*, *kEptpListAddress* and *kVmFunctionCtrl*) in the VmcsField to enable the function that reloading the EPT table in the non-root mode.

The other code including about 5,000 lines will be run mainly in the kernel context of the target system. Target processes and threads are tracked by the callback functions registered with the exported kernel functions. Once the target process is created, we allocate the resident memory space for storing the generated code. During the execution of target program, the functions of adjusting process working set are hooked inline to facilitate tracking the physical memory page allocation and release, then the permissions of physical pages related to target code in the EPT entries are set to non-execute, and the pages for storing the memory analysis state are also allocated. The entry and exit points of system calls are also intercepted to redirect the execution after returns to the user mode, although for non-open source systems, such interception approaches need to overcome some challenges, for example, version changes, kernel protections, and so on. In addition, we modify the IDT entries of each processor, register the new routine *KiTrap14* to handle virtualization exceptions, and hook the original routine *KiTrap0E*. In the #VE handing procedure, we disassemble and parse the target binary code, then construct the new execution and analysis code. The instruction *VMFUNC* and kernel function *KeIpiGenericCall* are used to implement the page access permission changes and the execution synchronization between different processors, when intercepting the code self-modifying.

IV. EXPERIMENTS AND RESULTS ANALYSIS

This article implements the prototype framework of the analysis method based on the 32-bit Windows platform, and uses various kinds of real programs to evaluate its performance and effectiveness. The main experimental environment is the common desktop computer, which is configured with Intel i5-7500 @3.40GHz 4 cores, 4 GB memory, 120 GB solid system disk and 500 GB data disk, and the operating system installed is 32-bit Windows 10 (10240).

A. INSTRUMENTATION PERFORMANCE EVALUATION

To evaluate the impact of mere dynamic instrumentation on the program execution, we choose Pin (3.11-979988) as a comparison. The proposed framework can be directly deployed on the target system to analyze the installed programs, which is similar to the *Pin* tool. When the target program is executed and instrumented by *Pin*, the trace granularity instrumentation is used to interpret and execute the program code [12], only the number of basic blocks encountered by Pin is counted in the callback registered by the interface *TRACE_AddInstrumentFunction*, and no actual analysis code is added into the target program. This also means that the target program is only executed under the *Pin* environment [35]. Likewise, the proposed framework also intercepts the target program execution and instruments the code based on the basic block granularity. The experimental purpose is to only evaluate the performance of the dynamic execution interception and code rewriting. In the experiment, first, the real programs are used for experiments, we choose various types of system built-in programs, and commonly used compression programs such as 7-Zip (18.05) and WinRAR (5.30), and the specific programs are shown in Fig. 9. Specifically, ce*rtutil* is used to calculate the sha256 hash value of the text file, and *wmic* is used to obtain the basic *bios* information in the experiment. Then *cscript* and *mshta* are used to download the file based on *vbscript* and *javascript* respectively, and a graphical window will briefly pop up during the execution of the latter program. To avoid the impact of code loop execution after the instrumentation is completed, the size of each file handled is set to 1 KB. In addition, except for two compression programs, others are command line programs. Each experiment scenario is repeated 10 times at least, and the average of the recorded data is token as the experimental result shown as Fig. 9 and Table 1.

The execution time shown in Fig. 9 is the increased value based on the native execution time of the target program. From the figure, it can be seen that the time overhead of instrumenting the target program by the proposed method is lower than *Pin* tool, which is presented in various types of programs. As shown in Table 1 and Fig. 9 together, with the increase of the number of basic blocks executed by the target program, the execution time increases gradually when *Pin* is used as the instrumentation tool, but the proposed method is less affected. A large amount of code loading in a short time could cause the time increase of instrumentation, and excessive time spending may lead to the execution failure of the delay sensitive code, for example, malware often uses time-based analysis detection techniques. For the analysis under the proposed framework, only a new basic block is executed by the program, the virtualization exception will be generated and handled in most cases, and the total number of exceptions is in a lower range, so the exception impact on the runtime performance is small. In fact, frequent virtualization exceptions can also have a significant impact on the execution of the target program, which has been studied and discussed in the previous work [31]. In Table 1, it can also be observed that although the target program creates many threads and loads a lot of modules during the execution, it does not result in a significant impact on the interception efficiency. Similarly, the amount of memory pages actually allocated by the program also keeps the overhead of memory used for storing the generated code and analysis state within a reasonable range. In addition, as shown in Table 1, the number of virtualization exceptions is slightly higher than the number of basic blocks, these extra exceptions are generated in the process of building and linking different generated code blocks, also including the interception that fails to cover all return points of system calls. In general, the interception solution proposed in this article has a small impact on the execution of the target program in most cases.

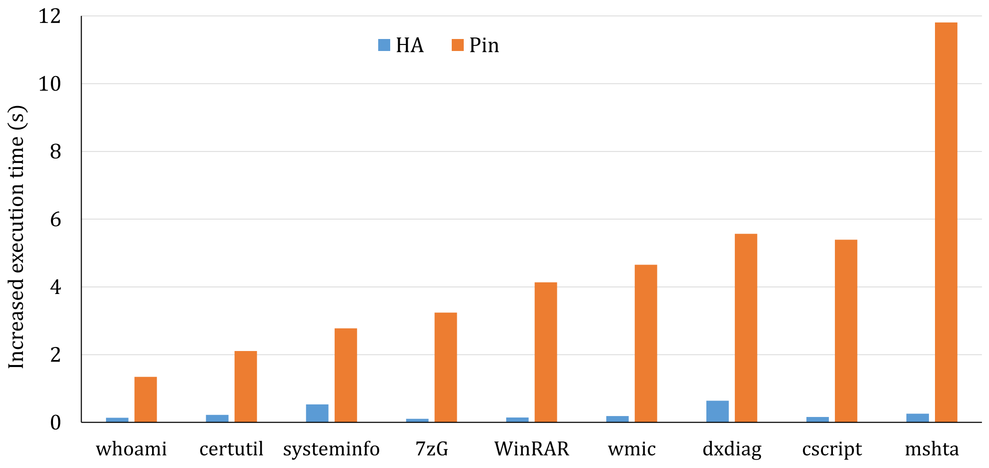


FIGURE 9. Increased execution time due to the dynamic instrumentation.

TABLE 1. Data statistics for the execution interception based on program basic blocks.

Program	Created threads	Loaded modules	Allocated pages	Executable code pages	Basic blocks (K)	#VE (K)
whoami	4	22	1050	653	21.0	26.0
certutil	4	53	1982	1177	31.6	40.5
systeminfo	7	32	1595	1030	45.7	58.8
7zG	6	30	2775	1447	56.2	71.6
WinRAR	39	39	4952	1726	72.4	94.8
wmic	7	46	3769	1810	83.6	104.3
dxdiag	11	78	3462	2054	95.4	141.6
cscript	11	61	3358	2208	98.6	123.1
mshta	15	69	5886	4049	188.8	235.5

Then we perform the same experiments on the SPEC CPU 2006 benchmark using the test workload, to avoid the impact of loop execution of the generated code by instrumentation. In the experiment, we exclude some programs that failed to compile successfully in the target environment, and a few programs that takes up a large amount of memory at runtime. As shown in Fig. 10, more results can be obtained, first, in most cases, HA has a high interception speed, and it also takes a little more time than *Pin* tool for a few programs. This is because these program code has fewer basic blocks and includes fewer instructions, so the instrumentation process can be done quickly, and then the program will execute mainly the existing code in a loop. For example, such as *458.sjeng*, it only generates about 10 thousand basic blocks and 50 thousand instructions in the execution process. And this also shows that the execution overhead of HA in terms of generating and linking code is more than *Pin* in some cases. In fact, the *Pin* tool spends more time to generate execution code and arrange its memory layout, and the goal is to achieve better performance in the code loop execution after the instrumentation is completed. For example, for *400.perlbench* and *445.gobmk*, the time overhead caused by the instrumentation is cumulative, because the test workload contains multiple program re-executions.

B. TAINT ANALYSIS EVALUATION

On the basis of above experiments, we further use multiple types of programs to comprehensively evaluate the performance and resource overhead of the instrumentation and specific analysis, first, the real programs are used which are shown in Fig. 10, which involve file compression, encryption, format conversion, and network communication. When testing, the target program is executed with the default parameter, and the size of the file handled is 1 MB of text file, moreover, a local server is built upon Python to avoid the impact of network speed. Compared with the last experiments, this time we add the tests of the built-in compression program *makecab* and the encryption program GnuPG (2.2.20), also use the third-party tool aria2c (1.34) for file downloading, and use the FFmpeg (4.0.2-static) tool to convert the wav file (Windows\Media\Ring05.wav) to the mp3 format. To avoid the impact of graphical interactions on *Pin*, the command line version of the target program is used here.

The experimental results are shown in Fig. 10, we calculate the program execution time of test scenarios separately, where HA indicates that the target program is only instrumented without inserting the analysis code, while the DTA suffix means that the dynamic taint analysis is synchronously performed based on the instrumentation. For the analysis with Pin, the taint analysis code is created based on the libdft (3.1415alpha) engine [17], which can be easier to migrate to the Windows platform. We compile the source code with VS2015, while using the byte-level analysis and the O3 optimization scheme. HA_DTA mea ns instrumenting and analyzing the same instructions, where some uncommonly used instructions are excluded on the basis of the original *libdft*, and the specific names of the instructions instrumented here are shown in Table 2. In the performance testing, the callbacks of target process creation and exit are token as the taint source and sink respectively, that is when the first instruction of the target program is executed, the instrumentation and taint analysis begins until the target process exits. From the experimental results it can be observed that the method proposed in this article can achieve high analytical efficiency in analyzing

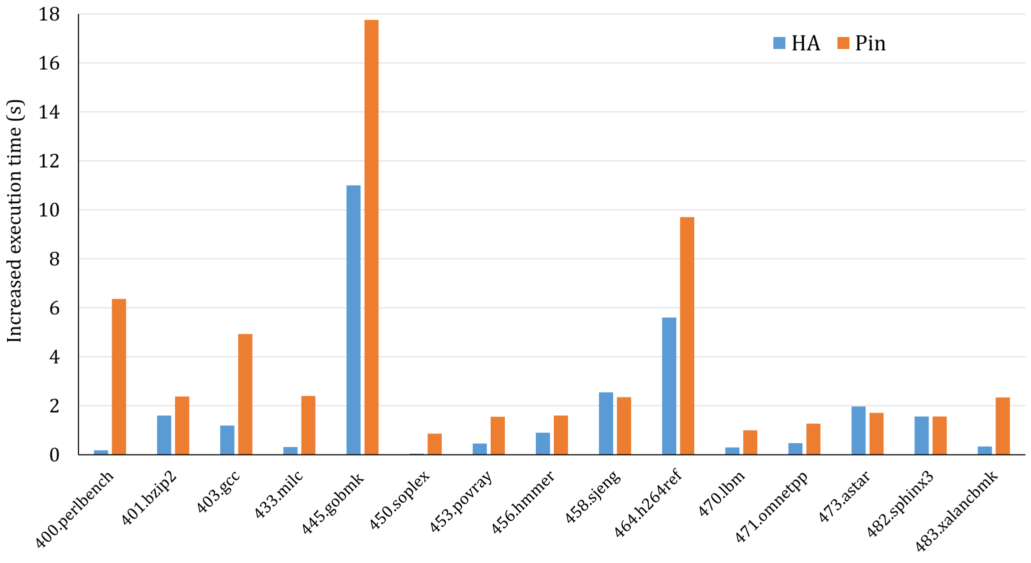


FIGURE 10. Increased execution time due to the dynamic instrumentation on the SPEC CPU 2006.

TABLE 2. Instrumented instructions for taint analysis in the experiment.

Instruction type	Mnemonic
Data Transfer Instructions	MOV, CMOVcc, MOVSX, MOVZX, POP, PUSH, XCHG, CMPXCHG, XADD
Binary Arithmetic Instructions	ADC, ADD, SBB, SUB, DIV, IDIV, MUL, IMUL
Logical Instructions	AND, OR, XOR
Bit and Byte Instructions	BSF, BSR, SETcc
String Instructions	LODSB, LODSW, LOWSD, STOSB, STOSW, STOSD, MOVSB, MOVSW, MOVSD
Other Instructions	LEA, PUSHFD, CPUID, CBW, CWD, CWDE, CDQ, STMXCSR, SMSW, LAHF, XLAT

TABLE 3. The experimental statistics of the analysis for different programs.

Program	Allocat ed Pages	Memory usage of code rewriting (MB)	Basi c bloc ks (K)	#VE (K)	Instructions	
					Parsed (K)	Executed (M)
7z	5910	5.7	23.5	31.7	112	1574.4
makecab	852	3.8	15.8	25.0	76	330.4
FFmpeg	2721	10.3	41.8	52.7	209	2385.1
Rar	13206	9.6	41.7	53.3	193	398.3
GnuPG	1789	8.6	34.2	43.1	169	593.6
certutil	2005	7.4	31.6	40.6	147	77.9
cscript	4191	23.1	98.9	129.4	446	47.5
aria2c	2423	11.7	45.4	60.6	222	19.2

various types of real programs, and the performance overhead of the target program under the analysis is very close to the native execution. Therefore, this method can be better used in real-time monitoring and analysis, and reduce the impact on the running program. In addition, the average CPU usage for various analyses is similar to that of traditional analysis, which would increase due to the introduction of additional analysis. On the one hand, it depends on the original CPU usage during the native execution of the program. On the other hand, it depends on the time spent on the analysis and the impact can be ignored when the analysis time is short.

The other statistics are shown in Table 3, as the size of handled file increases, the amount of memory dynamically allocated by the target program increases, and the memory space of storing the analysis state also increases, but the overall memory overhead is within an acceptable range. At the same time, the usage of memory that is used to store the generated code and build the specific analysis code is also within a reasonable range. Although the performance of analysis is affected by various factors, the proposed method still performs efficient taint analysis for different application scenarios in practice. In the analysis of different programs, a small number of exceptions are generated, which also leads to the lower impact on the whole system.

To further evaluate the performance impact of the compact inline analysis, we continue to perform the experiments with some of the test programs above, but the size of the file handled by each program is increased to 10 MB this time. In the experiment, the compression program WinZip (21.0) and the graphic conversion tool ImageMagick (7.0.10-dll) are further introduced, then we use the latter to convert the JPG image with the size mentioned above to the PNG format. In the same way, the native execution time and the analysis time of the target program are recorded separately. As shown in Fig. 12, compared with the traditional analysis, the compact inline analysis proposed here can still achieve high analytical performance with the growth of the handled file size. Similar to the traditional instrumentation, the inline analysis also leads the original code to expand, but the more compact analysis instructions are adopted to reduce the overhead of excessive context switches. Code instrumentation and memory allocation are performed in kernel space, which also reduces the

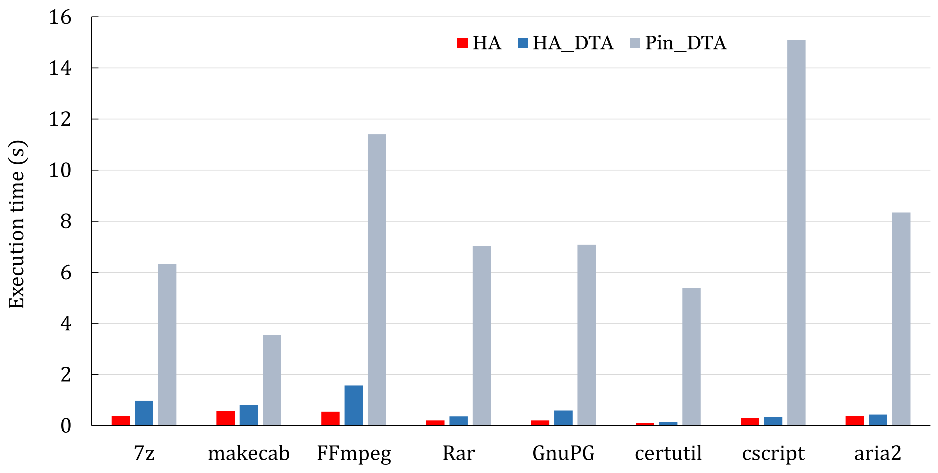


FIGURE 11. Analysis performance comparison of different programs.

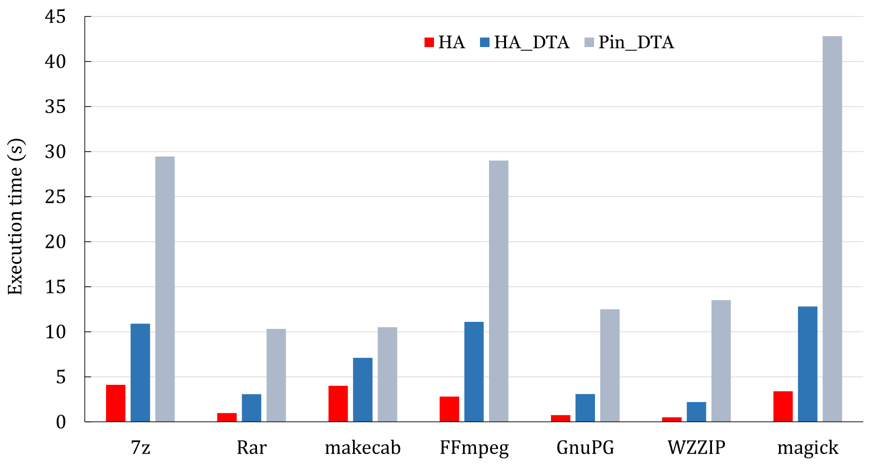


FIGURE 12. The analysis time when larger file is handled in the execution of different programs.

switches between user and kernel mode. So it can mitigate the runtime performance reduction of the target program while achieving the same analysis goal. In fact, the more times the generated code is executed, the greater the cumulative impact on the execution time of the program.

On the basis of real program testing, finally we continue using SPEC CPU 2006 to evaluate the performance of taint analysis, and use the ref workload this time. Unlike the actual program above, most benchmark programs do not involve file outputs, which can reduce the impact of disk I/O on processing large files. In addition, most benchmark programs also load less code which will soon be instrumented during the execution, and then enter the loop execution of the existing generated code. After the experiment is completed, the elapsed time of native program execution is used as the baseline to calculate the speed reduction ratio under the other analysis scenarios. As shown in Fig. 13, the experimental results are similar to the above tests, for the vast majority of programs, HA can achieve the better analysis performance than the traditional analysis based on Pin. Because it uses more streamlined instructions to complete the byte-level taint analysis, and reduces the amount of overhead generated by the execution switches between the analysis code and the native code in the traditional analysis, it is better to control the program performance reduction of the overall analysis in the long time execution. On the other hand, because HA is mainly located in kernel space, it is able to make full use of the advantages of large-page memory accesses. Since we do not analyze the floating point related instructions such as SSE, AVX, the analysis has a small effect on the execution of the program that focuses on floating point arithmetic (e.g. *433.milc*, *444.named*). It is also shown that the performance of HA will be slightly lower than the Pin tool for the analysis of the less instrumented program with long execution (e.g. 470.lbm), as discussed also in the last section.

C. CASE STUDY

1) ANALYSIS OF POWERSHELL SCRIPTS

The powerful PowerShell command-line environment has been integrated into the latest Windows system, and PowerShell scripts are widely used in system management and malicious programs. We use the proposed method to analyze the

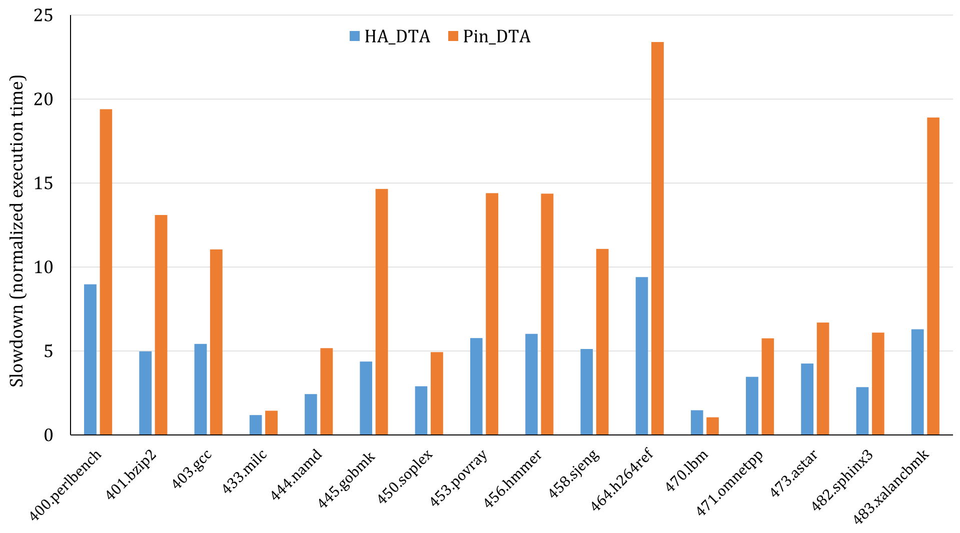
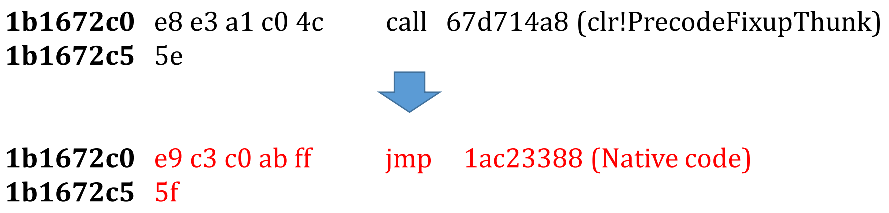


FIGURE 13. The execution slowdown for the analysis on SPEC CPU 2006.


```
1b1672c0  e8 e3 a1 c0 4c      call   67d714a8 (clr!PrecodeFixupThunk)
1b1672c5  5e

1b1672c0  e9 c3 c0 ab ff      jmp    1ac23388 (Native code)
1b1672c5  5f
```

FIGURE 14. Example of code modification in the execution.

execution of PowerShell scripts at the instruction level. In the experiment, the executed script uses the *System.net.webclient* object to download the text file of 1 KB from the local server. In analysis, the kernel functions such as *NtDeviceIoControlFile* and *NtWriteFile* are intercepted as the taint source and sink respectively. When the target data is received from the network, it is marked as tainted, and the analysis state of the data to be written will be checked in the file writing process. In multiple experiments, we can all succeed in capturing the data flow propagations, and the results show that this method can be applied to the analysis of complicated programs. The average time spent of the analysis process is about 15 s while the native execution of the script takes about 2.5 s, and some specific results of the experiment are shown in Table 3. As seen from the table, the program dynamically loads lots of modules and produces a large number of basic blocks during the execution. When analyzing the execution above based on the *Pin* tool, the entire analysis process takes nearly 100 s, which is too expensive.

Because the PowerShell environment is built on the .NET framework, it actually calls the .NET modules when the target script is interpreted and executed. While the .NET code is executed, the common language runtime (CLR) will immediately compile the intermediate code to the executable instructions on the target platform, so the PowerShell program will involve the modifications to the original binary code in the script execution process. An example is shown in Fig. 14, The JIT process is triggered when the function at 1b1672c0 is called first time, which will compile the intermediate code to the platform related executable code, while modifying the jump target at the beginning of the original function. The proposed method can also capture and handle the situation. As shown in Table 4, during the entire analysis, the number of modifications captured at page level is about 13 K times which results in the additional 92 K intercepts of instruction execution to check the code changes of the basic blocks located in the affected page, although only nearly 700 basic blocks have been actually changed. This could fully capture the changes of the code during the execution, and does not have a large impact on the analysis performance.

TABLE 4. Experimental statistics of file downloading analysis based on PowerShell script.

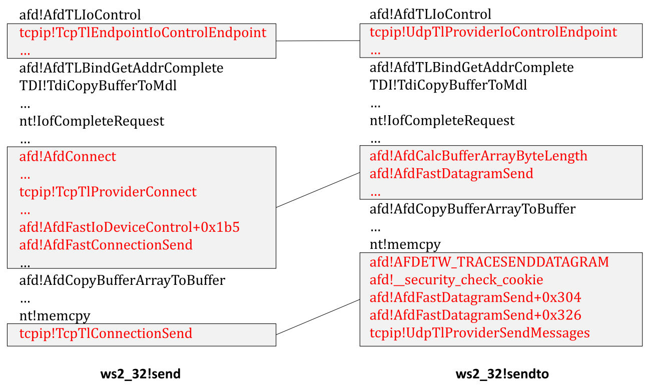
Created threads	Loaded modules	Basic blocks			#VE	
		Generated	Modified	Checked	Execute	Write
21	116	328K	694	92K	578K	13K

2) FUNCTION CALL ANALYSIS OF AFD DRIVER

On the Windows platform, the *AFD* driver is the ancillary driver that supports the Winsock function used by user mode, it means that some of the steps need to be handled by this driver, when using the socket function to send and receive data in user space. In this experiment, we write two basic socket programs, using the *send* and *sendto* function to send the TCP packets and UDP packets respectively, then try to explore the handling process for the two types of sending packet by the *AFD* driver.

In order to achieve this goal, we use the method of this article to instrument the test program in user space, and also instrument the code of *AFD* driver loaded in the kernel. When the generated and analysis code is built in kernel space, the context information is saved as needed through the kernel stack of the target thread. Because the driver code in kernel space is shared by all the applications and the operating system itself, and to avoid the impact on the system operation, we also replicate the physical pages related to the *AFD* driver and remap the virtual addresses in page table of the target process only. In fact, after the system boot is finished, the driver is loaded in the kernel, and 63 physical pages have actually been allocated for storing its code and data. In the modification of page table of the target process, we need to replicate the allocated original physical pages, replace the values of corresponding page table entries, and also replicate a page directory table that contains these page table entries above. After that, we further modify the access permissions in the EPT entries for the relevant replicated pages.

Through multiple experiments, it can be found that the performance impact on the running system is negligible for the analysis of packet sending in both cases. About 720 virtualization exceptions are generated by the code execution in kernel mode and about 650 basic blocks of *AFD* driver code are produced for the process of sending TCP packets, which is less for the UDP case. The number of exceptions generated is higher than the number of parsed basic blocks, because there is also the execution from the external kernel modules to the *AFD* driver, such as *tdi*, *tcpip*, and *nt*. Finally, we recorded the sequence of target addresses of *call* and *jmp* instructions generated in the execution, about 200 records are generated in the experiment for *send*, and about 160 for *sendto*. Then the two sequences are compared using the file comparison tool, and the partial differences are shown in Fig. 15, we can



```
afd!AfdTLIoControl
tcpip!TcpTlEndpointIoControlEndpoint
...
afd!AfdTLBindGetAddrComplete
TDI!TdiCopyBufferToMdl
...
nt!IofCompleteRequest
...
afd!AfdConnect
...
tcpip!TcpTlProviderConnect
...
afd!AfdFastIoDeviceControl+0x1b5
afd!AfdFastConnectionSend
...
afd!AfdCopyBufferArrayToBuffer
...
nt!memcpy
tcpip!TcpTlConnectionSend
```

ws2_32!send

```
afd!AfdTLIoControl
tcpip!UdpTlProviderIoControlEndpoint
...
afd!AfdTLBindGetAddrComplete
TDI!TdiCopyBufferToMdl
...
nt!IofCompleteRequest
...
afd!AfdCalcBufferArrayByteLength
afd!AfdFastDatagramSend
...
afd!AfdCopyBufferArrayToBuffer
...
nt!memcpy
afd!AFDETW_TRACESENDDATAGRAM
afd!__security_check_cookie
afd!AfdFastDatagramSend+0x304
afd!AfdFastDatagramSend+0x326
tcpip!UdpTlProviderSendMessages
```

ws2_32!sendto

FIGURE 15. Contrast of the call sequences for different sending functions, and the red part is the difference.

easily observe the difference between the *AFD* driver's handling processes for the two types of data sending, and can obtain the specific function names and the invoked functions in the external modules. From this case analysis, it can be seen that the proposed method can also be able to analyze the driver code in kernel space, which shows the good applicability.

V. RELATED WORK

The dynamic binary instrumentation and analysis method can be applied in many areas such as malicious code analysis, vulnerability analysis, software debugging, there exists a lot of research on dynamic analysis, and this article describes and discusses the related work mainly in two aspects.

A. ANALYSIS BASED ON APPLICATIONS

There are widely used instrumentation tools such as Pin [12], DynamoRIO [13], Frida [15], based on these tools various methods of data flow analysis or taint analysis can be quickly implemented [37]. The recently proposed method Instrew [40] uses LLVM to implement the binary code translation and analysis, which can further improve the analysis efficiency. Tinyinst [41] can be used to instrument and analyze part of the program, which is similar to the proposed method for this respect. The advantage of instrumentation in user mode is that it is relatively easy to implement the software based program dynamic translation, and can provide rich development interfaces that enable users to quickly implement the required analysis tools. The downside is that there is a lack of sufficient capabilities to counter the anti-analysis techniques used by malware in user mode, and it also results in lower analysis performance in some cases. In comparison, the core part of proposed framework is located in kernel space, which will be more flexible in coping with the analysis detection techniques, and it can achieve the higher instrumentation performance in most cases. In addition, to further improve the analysis efficiency on the foundation of instrumentation, more efficiency specific binary analysis methods have been proposed. On the one hand, the execution of the original program is decoupled from the analysis procedure, such as ShadowReplica [33], on the other hand, the online information recording is combined with the further offline analysis, such as StraightTaint [35], and most of these studies are still based on the traditional instrumentation methods. In this article, a new dynamic instrumentation solution and the matching online taint analysis method are proposed, based on which other specific analysis methods can also be applied.

B. ANALYSIS BASED ON VIRTUALIZATION AND HARDWARE

To improve the scope and stealthiness of the analysis, more and more studies are using virtualization methods and hardware features [5], [8]. The behaviors of malicious programs can be analyzed within the system range, such as DRAKVUF [11], Panorama [28]. In the aspect of virtualization, the framework DECAF is implemented based on the QEMU emulator [10], [29], which can also support the fine-grained analysis. Ether [32] is also a framework for analyzing the whole system based on the customized hypervisor. These studies can further improve the capability of program analysis, but also have difficulties in the deployment and flexibility. SPIDER [26] and MALT [27] mainly focus on improving the transparency of debugging and analysis, but there are challenges in the aspects of semantic information acquisition and automated analysis. Both PEMU [24] and PinOS [25] are the analysis frameworks that extend the Pin tool to support the analysis of the whole system, but the issues such as availability, performance, and semantic gaps still exist. In comparison, this article intercepts the execution and analyzes the target program based on the virtualization exception and system kernel, which has better performance and flexibility, and can perform the program analysis by directly loading the kernel module on the target system. It can directly obtains the semantic information of target program at runtime, and achieve the better balance in terms of availability and stealthiness. In addition, there also exists some record and replay systems [38], [39], which are essentially the offline analysis methods based on the runtime recording by the code instrumentation.

VI. DISCUSSION

This article presents the new program instrumentation and analysis method, and implements the preliminary prototype framework. Compared with the existing mature tools, although it has better analytical performance, flexibility and applicability in some analysis scenarios, there are still some imperfect places currently, for example, not providing rich user development interfaces, lack of a good dynamic memory management and garbage collection mechanism, non-optimized memory layout of the generated code, only supporting the byte-level analysis currently, etc., which can be improved in further work referring to the existing analysis tools and methods. This method has certain universality and can be complemented with mature analysis tools to accomplish the analysis work in the cases of delay sensitive code and kernel modules.

In response to the anti-analysis and detection adopted by the target program, similar to the traditional analysis methods, the proposed method still needs to face many challenges. However, compared with the analysis method in the application layer, it will be easier and effective to adopt countermeasures in kernel space, and can use the protection mechanism based on the virtualization techniques. Since the target operating system is directly converted to run in the virtualized mode while analyzing, the method still has a certain advantage in the aspects of the hidden of virtualization characteristics and the usability, compared with the methods based on the emulator and virtual machine.

This article presents a new solution for dynamic program execution interception, it can rewrite part of the program, and leave other code to run in native mode. In addition, the proposed solution could also be implemented through modifying the permissions of page table entries in the kernel, but it causes more interference in memory management of the operating system and lacks stability to some extent.

At present, this article implements the framework on the 32-bit platform, and performs the relevant experiments. In fact, when the target program takes up a great deal of memory, it will lead to a lack of memory addresses for analysis, which will be improved on the 64-bit platform. Next, the experiment needs to be extended to the 64-bit environment and the latest hardware platform. In addition, the method proposed could be easy to achieve in other platforms, but it should be further strengthened in the perfection of design and the stability of prototype implementation.

VII. CONCLUSION

This article proposes a new dynamic instrumentation and analysis method for binary programs, which can accomplish automatic fine-grained analysis of the target program and its kernel modules based on the new virtualization mechanism. The architecture of this method is simple, which is easy to implement and apply in the actual analysis scenario quickly. Lots of real experiments have shown that the method is effective, and has high analytical performance and applicability.

At present, we implement the prototype of the method on the Windows platform and verify its feasibility, but there still exist shortcomings. Next, we will improve the design and implementation of the framework, introduce the user mode interface, and enhance the ease of use. Assembly tools can be introduced to increase the automation of the analysis code generation, so that other analysis functions can be quickly implemented. We could optimize the method of code generation in the instrumentation and perform experiments on the 64-bit platform, in order to further enhance the application capability in actual analysis.

CONFLICT OF INTEREST

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this article.

REFERENCES

[1] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, “A survey on automated dynamic malware-analysis techniques and tools,” *ACM Comput. Surv.*, vol. 44, no. 2, pp. 1–42, Feb. 2012.

[2] M. Zhang, R. Qiao, N. Hasabnis, and R. Sekar, “A platform for secure static binary instrumentation,” in *Proc. 10th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environ. (VEE)*. New York, NY, USA: Association Computing Machinery, 2014, pp. 129–140.

[3] G. J. Duck, X. Gao, and A. Roychoudhury, “Binary rewriting without control flow recovery,” in *Proc. 41st ACM SIGPLAN Conf. Program. Lang. Design Implement.*, London, U.K., Jun. 2020, pp. 151–163.

[4] D. C. D’Elia, E. Coppa, S. Nicchi, F. Palmaro, and L. Cavallaro, “SoK: Using dynamic binary instrumentation for security (and how you May get caught red handed),” in *Proc. ACM Asia Conf. Comput. Commun. Secur.* New York, NY, USA: Association Computing Machinery, Jul. 2019, pp. 15–27.

[5] E. Bauman, G. Ayoade, and Z. Lin, "A survey on hypervisor-based monitoring: Approaches, applications, and evolutions," *ACM Comput. Surv.*, vol. 48, no. 1, pp. 1–33, Sep. 2015.

[6] M. Polino, A. Continella, S. Mariani, S. D'Alessio, L. Fontana, F. Gritti, and S. Zanero, "Measuring and defeating anti-instrumentation-equipped malware," in *Proc. 14th Conf. Detection Intrusions Malware, Vulnerability Assessment (DIMVA)*, Bonn, Germany, Jul. 2017, pp. 73–96.

[7] D. C. D'Elia, E. Coppa, F. Palmaro, and L. Cavallaro, "On the dissection of evasive malware," *IEEE Trans. Inf. Forensics Security*, vol. 15, pp. 2750–2765, 2020.

[8] 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 *Proc. 28th Annu. Comput. Secur. Appl. Conf. (ACSAC)*, Orlando, FL, USA, 2012, pp. 189–198.

[9] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to Ask)," in *Proc. IEEE Symp. Secur. Privacy*, Oakland, CA, USA, May 2010, pp. 317–331.

[10] 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 Trans. Softw. Eng.*, vol. 43, no. 2, pp. 164–184, Feb. 2017.

[11] T. K. Lengyel, S. Maresca, B. D. Payne, G. D. Webster, S. Vogl, and A. Kiayias, "Scalability, fidelity and stealth in the DRAKVUF dynamic malware analysis system," in *Proc. 30th Annu. Comput. Secur. Appl. Conf. (ACSAC)*, New Orleans, LA, USA, 2014, pp. 386–395.

[12] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement.* New York, NY, USA: Association Computing Machinery, 2005, pp. 190–200.

[13] D. Bruening, Q. Zhao, and S. Amarasinghe, "Transparent dynamic instrumentation," in *Proc. 8th ACM SIGPLAN/SIGOPS Conf. Virtual Execution Environ. (VEE)*. New York, NY, USA: Association Computing Machinery, 2012, pp. 133–144.

[14] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proc. 28th ACM SIGPLAN Conf. Program. Lang. Design Implement.* New York, NY, USA: Association Computing Machinery, 2007, pp. 89–100.

[15] Frida. *Dynamic Instrumentation Toolkit*. Accessed: Jun. 20, 2020. [Online]. Available: https://frida.re/

[16] QBDI. *QuarkslaB Dynamic Binary Instrumentation*. Accessed: Jun. 20, 2020. [Online]. Available: https://qbdi.quarkslab.com/

[17] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, "Libdft: Practical dynamic data flow tracking for commodity systems," *ACM SIGPLAN Notices*, vol. 47, no. 7, pp. 121–132, Sep. 2012.

[18] J. Ming, D. Xu, L. Wang, and D. Wu, "LOOP: Logic-oriented opaque predicate detection in obfuscated binary code," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*. New York, NY, USA: Association Computing Machinery, 2015, pp. 757–768.

[19] A. Prakash, X. Hu, and H. Yin, "VfGuard: Strict protection for virtual function calls in COTS C++ binaries," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, San Diego, CA, USA, 2015, pp. 1–15.

[20] J. Li, Z. Lin, J. Caballero, Y. Zhang, and D. Gu, "K-hunt: Pinpointing insecure cryptographic keys from execution traces," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, New York, NY, USA, Jan. 2018, pp. 412–425.

[21] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su, "X-force: Force-executing binary programs for security applications," in *Proc. USENIX Secur. Symp. (USENIX Secur.)*, San Diego, CA, USA, Aug. 2014, pp. 829–844.

[22] E. Zhu, P. Wen, K. Ni, and R. Ma, "Implementation of an effective dynamic concolic execution framework for analyzing binary programs," *Comput. Secur.*, vol. 86, pp. 1–27, Sep. 2019.

[23] J. Kirsch, Z. Zhechev, B. Bierbaumer, and T. Kittel, "PwIN—Pwning Intel piN: Why DBI is unsuitable for security applications," in *Computer Security*. Barcelona, Spain: Springer, Sep. 2018.

[24] J. Zeng, Y. Fu, and Z. Lin, "PEMU: A pin highly compatible Out-of-VM dynamic binary instrumentation framework," in *Proc. 11th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environ. (VEE)*, New York, NY, USA, 2015, pp. 147–160.

[25] P. P. Bungale and C.-K. Luk, "PinOS: A programmable framework for whole-system dynamic instrumentation," in *Proc. 3rd Int. Conf. Virtual Execution Environ. (VEE)*. New York, NY, USA: Association Computing Machinery, 2007, pp. 137–147.

[26] Z. Deng, X. Zhang, and D. Xu, "SPIDER: Stealthy binary program instrumentation and debugging via hardware virtualization," in *Proc. 29th Annu. Comput. Secur. Appl. Conf. (ACSAC)*, New Orleans, LA, USA, 2013, pp. 289–298.

[27] F. Zhang, K. Leach, A. Stavrou, H. Wang, and K. Sun, "Using hardware features for increased debugging transparency," in *Proc. IEEE Symp. Secur. Privacy*, San Jose, CA, USA, May 2015, pp. 55–69.

[28] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: Capturing system-wide information flow for malware detection and analysis," in *Proc. 14th ACM Conf. Comput. Commun. Secur. (CCS)*, Alexandria, VA, USA, 2007, pp. 116–127.

[29] A. Davanian, Z. Qi, Y. Qu, and H. Yin, "DECAF++: Elastic whole-system dynamic taint analysis," in *Proc. 22nd Int. Symp. Res. Attacks, Intrusions Defenses*, Beijing, China, Sep. 2019, pp. 31–45.

[30] S. Proskurin, M. Momeu, S. Ghavamnia, V. P. Kemerlis, and M. Polychronakis, "XMP: Selective memory protection for kernel and user space," in *Proc. IEEE Symp. Secur. Privacy (SP)*, San Francisco, CA, USA, May 2020, pp. 603–617.

[31] J. Pan, Y. Zhuang, and B. Sun, "BAHK: Flexible automated binary analysis method with the assistance of hardware and system kernel," *Secur. Commun. Netw.*, vol. 2020, pp. 1–19, Jan. 2020, doi: 10.1155/2020/8702017.

[32] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: Malware analysis via hardware virtualization extensions," in *Proc. 15th ACM Conf. Comput. Commun. Secur. (CCS)*, Alexandria, VA, USA, 2008, pp. 51–62.

[33] K. Jee, V. P. Kemerlis, A. D. Keromytis, and G. Portokalidis, "ShadowReplica: Efficient parallelization of dynamic data flow tracking," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, Berlin, Germany, 2013, pp. 235–246.

[34] 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," *Comput. Secur.*, vol. 51, pp. 1–15, Jun. 2015.

[35] J. Ming, D. Wu, J. Wang, G. Xiao, and P. Liu, "StraightTaint: Decoupled offline symbolic taint analysis," in *Proc. 31st IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Singapore, 2016, pp. 308–319.

[36] C. Wang and S. W. Shieh, "SWIFT: Decoupled system-wide information flow tracking and its optimizations," *J. Inf. Sci. Eng.*, vol. 31, no. 4, pp. 1413–1429, 2015.

[37] E. Zhu, F. Liu, Z. Wang, A. Liang, Y. Zhang, X. Li, and X. Li, "Dytaint: The implementation of a novel lightweight 3-state dynamic taint analysis framework for x86 binary programs," *Comput. Secur.*, vol. 52, pp. 51–69, Jul. 2015.

[38] B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan, "Repeatable reverse engineering with PANDA," in *Proc. 5th Program Protection Reverse Eng. Workshop*, Los Angeles, CA, USA, 2015, pp. 1–11.

[39] Y. Ji, S. Lee, M. Fazzini, J. Allen, E. Downing, T. Kim, A. Orso, and W. Lee, "Enabling refinable cross-host attack investigation with efficient data flow tagging and tracking," in *Proc. 27th USENIX Secur. Symp.*, Santa Clara, CA, USA, Aug. 2018, pp. 1705–1722.

[40] A. Engelke and M. Schulz, "Instrew: Leveraging LLVM for high performance dynamic binary instrumentation," in *Proc. 16th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environ.* New York, NY, USA: Association Computing Machinery, Mar. 2020, pp. 172–184.

[41] TinyInst. *A Lightweight Dynamic Instrumentation Library*. Accessed: Jun. 20, 2020. [Online]. Available: https://github.com/googleprojectzero/TinyInst

[42] HyperPlatform. *Intel VT-X Based Hypervisor on Windows*. Accessed: Jun. 20, 2020. [Online]. Available: https://github.com/tandasat/HyperPlatform

[43] udis86. *Udis86 1.7.2 Documentation*. Accessed: Jun. 20, 2020. [Online]. Available: http://udis86.sourceforge.net/manual/libudis86.html

JIAYE PAN received the master's degree in computer application technology from the Nanjing University of Aeronautics and Astronautics, Nanjing, China, in 2010, where he is currently pursuing the Ph.D. degree with the College of Computer Science and Technology. His research interests include system security, software security, and network security.

ZHUANG YI graduated from the Department of Computer Science, Nanjing University of Aeronautics and Astronautics, Nanjing, China, in 1981. She is currently a Professor and a Ph.D. Supervisor with the College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics. Her research interests include network distributed computing, information security, and dependable computing.

ZHAO XUE-JIAN received the M.Sc. and Ph.D. degrees in computer application technology from the Nanjing University of Aeronautics and Astronautics, Nanjing, China, in 2007 and 2011, respectively. He is currently an Associate Professor with the School of Modern Posts, Nanjing University of Posts and Telecommunications, Nanjing. His current research interests include wireless sensor networks, ad hoc networks, and big data.

BINGLIN SUN received the master's degree in computer science from the Nanjing University of Aeronautics and Astronautics, Nanjing, China, in 2019. His research interests include information security and reverse engineering.