Fine-Grained Binary Analysis Method for Privacy Leakage Detection on the Cloud Platform

Jiaye Pan[1], Yi Zhuang[1, *], Xinwen Hu[1, 2] and Wenbing Zhao[3]

Abstract: Nowadays cloud architecture is widely applied on the internet. New malware aiming at the privacy data stealing or crypto currency mining is threatening the security of cloud platforms. In view of the problems with existing application behavior monitoring methods such as coarse-grained analysis, high performance overhead and lack of applicability, this paper proposes a new fine-grained binary program monitoring and analysis method based on multiple system level components, which is used to detect the possible privacy leakage of applications installed on cloud platforms. It can be used online in cloud platform environments for fine-grained automated analysis of target programs, ensuring the stability and continuity of program execution. We combine the external interception and internal instrumentation and design a variety of optimization schemes to further reduce the impact of fine-grained analysis on the performance of target programs, enabling it to be employed in actual environments. The experimental results show that the proposed method is feasible and can achieve the acceptable analysis performance while consuming a small amount of system resources. The optimization schemes can go beyond traditional dynamic instrumentation methods with better analytical performance and can be more applicable to online analysis on cloud platforms.

Keywords: Cloud platform, privacy leakage, binary analysis, dynamic analysis.

1 Introduction

Cloud computing technology has been widely used in various areas, such as cloud storage, cloud server and website hosting [Liu, Liang, Susilo et al. (2016); Patel, Taghavi, Bakhtiyari et al. (2013)]. Meanwhile, as most cloud platforms are deployed in public network environments, security challenges have also been highly concerned and studied, such as data storage protection, access control, identification authentication and the running status monitoring [Liu, Liang, Susilo et al. (2016); Liu and Xia (2019); Rocha, Gross and

[1] College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, 200016, China.

[2] School of Computer Science and Engineering, Nanyang Technological University, 639798, Singapore.

[3] Department of Electrical Engineering and Computer Science, Cleveland State University, Cleveland, 44115-2214, USA.

[*] Corresponding Author: Yi Zhuang. Email: zy16@nuaa.edu.cn.

Received: 22 January 2020; Accepted: 01 April 2020.

Moorsel (2013)].

In most cases, cloud platforms are built using virtualization technology, that is, clients provided to users exist in the form of virtual machines (VMs). Multiple separated virtual machines share the same hardware resources of host machine, which are all supervised by the hypervisor or virtual machine monitor (VMM). In this situation, The more common threat is that malicious code existing in the virtual machine leads to user data stealing or turns the virtual machine into a mining node [Korkin and Tanda (2017); Patel, Taghavi, Bakhtiyari et al. (2013)]. Similarly, some normal cloud applications in virtual machines can also be infected or misconfigured, thus causing abnormal operation or information leakage. In view of this situation, the existing methods mostly focus on monitoring the behavior of programs, such as access operations related to processes, files and networks, including monitoring the inside and outside of virtual machines [Bauman, Ayoade and Lin (2015); Gupta and Kumar (2015); Han, Hao and Cui (2016); Mishraa, Pillia, Varadharajanb et al. (2017)]. However, it is difficult to detect and find more detailed abnormal operation of programs in real time. On the basis of previous research work, this paper proposes a real-time program monitoring and analysis method at instruction level to detect the possible data leakage in time.

There are some mature representative methods and tools in the field of traditional binary analysis, such as Pin [Luk, Cohn and Muth (2005)], but these methods or tools are not suitable for the analysis of running program or online monitoring on cloud platforms. In addition, although some research works based on virtual machines implement a system-wide program analysis system [Henderson, Prakash, Yan et al. (2017); Lengyel, Maresca, Payne et al. (2014)], including the fine-grained analysis, most involve static semantic analysis and depend on the instruction interpretation, and there are also challenges related to analysis granularity, efficiency, and deployment on cloud platforms.

To develop more applicable and lightweight analysis methods for the cloud applications, based on the previous research work, this paper proposes a new automatic privacy leakage monitoring method on cloud platforms, named as FBAC, which can perform online fine-grained binary analysis and reduce the performance impact on target programs. The method continues to employ the new features of virtualization technology, combines the extended page table (EPT) and virtualization exception (#VE) mechanisms to externally intercept the execution of target program with low performance overhead, this solution can also be feasible on the cloud architecture. Based on program behavior interception and fine-grained execution interception, all execution instructions of the target program are analyzed. Moreover, we propose the corresponding analysis optimizations to further reduce the performance overhead, which can meet the requirements of different analysis scenarios. So the proposed analysis method can be used to conduct dynamic taint analysis on target programs online to track the sensitive data propagations on cloud platforms.

Compared with other methods, the proposed method adopts online analysis at instruction level and is oriented to the whole program execution process. And by introducing more system level components based on the traditional hypervisor to improve analysis efficiency, which can also aid to accurately acquire the program semantic information. Moreover, along with internal binary instrumentation, optimization measures are proposed to overcome the performance bottleneck of automatic analysis in traditional pure virtualization mode. In addition, it leads to the lower impact on the target program and achieves the better performance. It can also accomplish the analysis that traditional instrumentation methods are difficult to be applied on cloud platforms. The contributions of this work are presented as follows:

Firstly, we propose a new online privacy data leakage detection method on cloud platforms, which can perform fine-grained automatic binary analysis. It can online monitor and analyze the suspicious program stably and efficiently based on multiple underlying components.

Secondly, we integrate the external execution interception with the internal instruction instrumentation and propose the corresponding analysis optimizations to further reduce the performance overhead caused by virtualization technology. Thus it enables real-time dynamic data flow analysis of target programs.

Finally, a prototype is implemented on the Windows and Intel platform, and multiple types of programs in actual environments, especially some network applications, are used to measure the performance and functionality. The results show that the method can be applied to online privacy leakage detection on cloud platforms.

2 Related works

There has been much research work related to cloud computing security, most focusing on data encryption protection, access control and operation resource monitoring [Liu, Liang, Susilo et al. (2016); Mishraa, Pillia, Varadharajanb et al. (2017)]. For example, Zhang et al. proposed the flexible monitoring framework for security status of virtual machine [Zhang and Lee (2018)], which provides a new VM security verification method and can automatically prevent violations of security rules. Fu et al. designed a distributed cloud application anomaly detection system FlowBox [Fu, Kim and Prior (2015)], which detects the performance anomaly behavior of cloud applications by collecting traffic information of multiple components. Rocha et al. proposes an attack model towards Xen platform [Rocha, Gross and Moorsel (2013)], which can break through the memory isolation and the corresponding protection model was also proposed. Lee et al. designed the protection system against the ransomware, which deployed the cloud system to collect and analyze all kinds of information for in-depth protection [Lee, Moon and Park (2017)]. Chung et al. [Chung, Khatkar, Xing et al. (2013)] proposed the new intrusion detection system for the cloud platform NICE, which can improve the detection capability by reconfiguring the virtual network structure. Most of the above studies protect the security of cloud platforms at macro level.

Researches regarding VM security monitoring are mainly based on hardware characteristics and VM introspection technology [Deng, Zhang and Xu (2013); Dinaburg, Royal, Sharif et al. (2008); Gupta and Kumar (2015); Mishraa, Pillia, Varadharajanb et al. (2017); Vogl and Eckert (2012); Willems, Hund, Fobian et al. (2012)]. However, some of the above researches are difficult to be deployed on cloud platforms, most of which are coarse-grained analysis work, or fail to realize efficient automatic analysis of the whole program, and may confront serious performance challenges during real-time online analysis on a large number of codes of the program. Besides, the binary analysis method at instrumentation level has been extensively studied [Jee, Kemerlis, Keromytis et al. (2013); Kemerlis, Portokalidis, Jee et al. (2012); Ming, Wu, Xiao et al. (2015); Schwartz, Avgerinos and Brumley (2010)]. Most of these synchronous researches involve modification and interpretation of target codes, which are difficult to be directly applied to online monitoring and analysis in cloud computing environments. However, the ideas involved in these researches are conducive to instruction-level program analysis on cloud platforms.

3 Method description

The overall architecture of the proposed method is shown as Fig. 1. It intercepts all memory accesses of the target running program first, including data acquisition and instruction fetching, and analyzes all the instructions that have been executed in the meanwhile. The scheduler component located in the hypervisor is responsible for controlling the code coverage and time duration of the analysis. The main components involved in each VM are system monitor and target program execution analysis component. The former monitors multiple types of program behaviors such as reading files and sending network data through system exported functions or function hooks. With this help, we can recognize the suspicious programs, mark the sensitive data in memory and detect the possible privacy leakage in the network traffic. Further the fine-grained analysis can be performed to determine if there exists the privacy data leakage, which is mostly conducted by the taint analysis engine. The execution analyzer traces and performs all kinds of analysis at the instruction level, also interacts with other components.

In virtualization mode, the virtual address and memory management mechanism of guest operating systems and applications will not be affected because EPT is used to translate the physical address of VM to real physical memory. Since EPT exceptions are only related to the current processor core, analysis among multiple clients will not affect each other. In addition, the fine-grained analysis can be launched timely and the code coverage can be dynamically adjusted, making the method more flexible. In the process of monitoring the client, the interception engine starts performing the coarse-grained monitoring functions, for example, obtaining the process list of target system, monitoring the use of system resources, intercepting the execution of critical system calls. Then fine-grained analysis is conducted for suspicious targets. Once the analysis is completed, fine-grained analysis should be closed in time to reduce the impact on the performance of the target system. Periodic fine-grained monitoring and analysis can also be directly carried out for the preset monitoring target. The execution state of target programs in the analysis and native state can be quickly and safely switched by changing the EPT pointer, and the state transition can be performed step by step by adjusting the permissions in the EPT table.

3.1 Design of key components

Two main parts are located in the guest OS kernel and hypervisor respectively. There is an analysis agent module in the kernel of each guest system. Since our main target is the code running in user mode, the analysis would mostly occur in kernel space, thus improving analysis efficiency. In kernel space of guest OS, a plurality of target system behavior monitors are constructed in the kernel module for monitoring program behaviors at the function level, which also interact with the component in charge of fine-grained data flow analysis. The components in the hypervisor schedule the analysis tasks, communicate with modules in the guest VM, construct the EPT structures and process the special VM exit events.

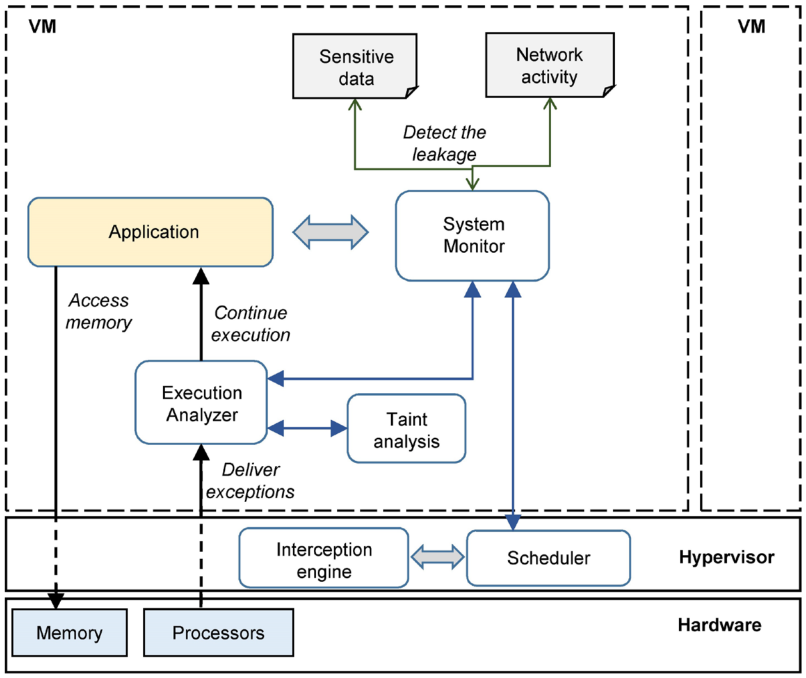


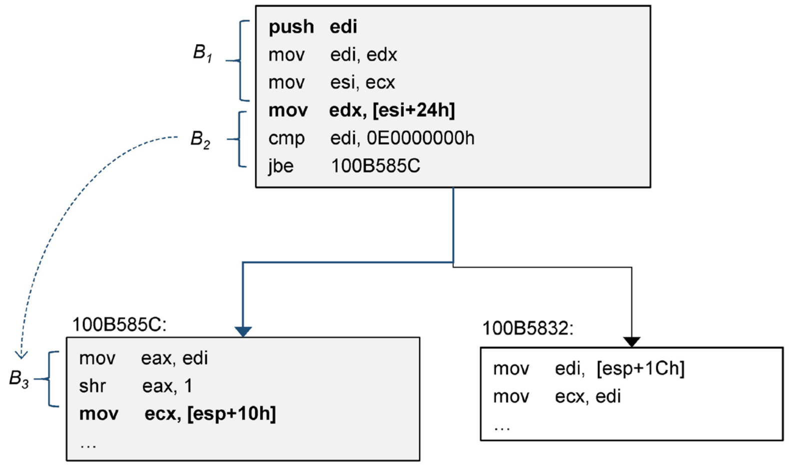
Figure 1: Overall architecture of FBAC

For each guest VM, multiple EPT tables need to be pre-constructed and shared by all processors of the VM. Each EPT table will be loaded in different analysis scenarios. For example, intercepting the program to be analyzed, or aiding to trace the executed instructions. The interception engine conducts the specific analysis, intercepts the execution of target program by setting access permissions of EPT entries, and also injects extra code into the guest machine according to the analysis requirement. It also schedules multiple analysis tasks in different guest VMs. EPT has supported EPT pointer (EPTP) switching mechanism since 6th Generation Intel Processor architecture Skylake, allowing to switch the EPT pointer of current processor in the guest VM without causing any VM exit. This new mechanism can help us to build more efficient interception solution [Pan, Zhuang and Sun (2020)].

The interception scheme should maintain the control of target program execution in the automated analysis, so during the handling of each EPT violation, also called EPT fault in this paper, we can directly emulator the memory access operation in kernel mode for most simply instructions, and the security check should be prior. For the instruction that is hard to be emulated, it can be executed in single-step mode as an option, then the execution will be intercepted in the subsequent debug exception handler. For the consideration of efficiency, new monitor trap flag is not used here which still causes the VM exit. In addition, the execution of target program generating #VE violation or exception will be profiled at the same time, as a guide for subsequent optimizations. In the optimization, the original code page will be rewritten and its subsequent execution will be redirected, so the interception of the code execution will be conducted in the inline form.

3.2 Instruction analysis

When abnormal behaviors of target programs are detected, then the monitoring of the program will shift to fine-grained mode, which means that the instruction level analysis will be performed on target programs. In this situation, we should trace and analyze all the instructions have been executed by the program to obtain the sensitive data propagation information and detect the possible privacy leakage before the entry points of important system calls. Because the read and write operations in memory of target programs are intercepted, the accessed memory addresses and corresponding memory values can be obtained, and other non-access memory instructions of the current thread can be analyzed through instant backtracking. For these instructions that access memory can be directly analyzed while handling the EPT fault. For other instructions, the runtime information and actual execution path can be further obtained by backtracking analysis, as shown in Fig. 2. Meanwhile the inline data flow analysis is also performed. Actually, for simple taint analysis, the inline analysis code only propagates the taint state between different memory addresses that store the analysis state of registers in the thread context.



```
push  edi
mov   edi, edx
mov   esi, ecx
mov   edx, [esi+24h]
cmp   edi, 0E0000000h
jbe   100B585C
```

```
100B585C:
mov   eax, edi
shr   eax, 1
mov   ecx, [esp+10h]
…
```

```
100B5832:
mov   edi, [esp+1Ch]
mov   ecx, edi
…
```

Figure 2: Example of tracing the instructions executed

Definition 1. Basic Analysis Block. It is a sequence of instructions in a program. As the traditional program basic block, any execution of the sequence starts from the first instruction of that and exits at the last. But the difference is that the first or the last instruction of basic analysis block can also be the one generating the EPT fault or exception. For the basic analysis block $B = \{c_1, \dots, c_k\}, k > 0$, it contains k instructions, where c_1 is the first instruction of traditional basic block or an instruction generating the fault, similarly, c_k is a branch instruction or also the instruction generating the fault. Moreover, one traditional basic block can include several basic analysis blocks, such as B_1 and B_2 shown in Fig. 2.

Definition 2. Local Analysis Path. It is a non-empty ordered set composed of basic analysis blocks, including all the instructions to be analyzed when the EPT fault occurs. The union of local analysis paths contains all the instructions executed by the program. For path $P = \{B_1, B_2, \dots, B_k\}, k > 0$, its length is k, where B_1 contains the instruction generating the last fault and B_k contains the one that generates the fault this time. The execution of instructions contained in $\bigcup_{i=2}^{k-1} B_i, k > 2$ do not cause any fault.

During the execution and analysis of target program, if addresses of two instructions that generate the consecutive EPT violations are the same or adjacent, there is no need for additional analysis of other executed instructions. Otherwise, if two instructions are located in the same basic analysis block, then it only needs to simply analyze the instructions between them, as the segment B_1 shown in Fig. 2.

If the instructions generating the exceptions for two consecutive times are not in the same basic analysis block, as shown in Fig. 2, B_2 and B_3 compose a local analysis path at the moment that the instruction "*mov ecx, [esp + 10h]*" is executed. In this situation, the branch information of program execution is needed to determine the accurate local analysis path. Therefore, we perform the fine-grained analysis and local re-execution synchronously to determine the actual analysis path between the last two faults. Since the instructions involving memory accesses have been directly intercepted, and the re-execution mainly involves register operations. In addition, when each exception occurs, the context information of the current thread is simultaneously saved for the initial environment during the next analysis, which can ensure the accuracy of the whole process and reduce error accumulation.

In the backtracking analysis, we progressively construct and cache the information of basic analysis blocks first. For the sake of simplicity, we adopt a two-level mapping structure to map the start address of each analysis block to its parsed information. For the 32-bit address in user mode, high 20 bits will serve as the index of the first-level mapping, so 2 MB of memory is required to store the mapping structures. The memory occupied by the second-level mapping depends on the number of code pages loaded by the running program, where each code page contains 4 K addresses and needs 16 KB additional memory space to store mapping structures. Subsequently, for each thread, we analyze the instructions of basic analysis block in the inline form, and decide whether to proceed to the analysis of next basic analysis block according to the analysis result, as shown in Fig. 3. Actually, the runtime values of registers are not needed for the instructions that do not involve memory accesses for traditional dynamic taint analysis. In this case, the re-execution with inline analysis for these instructions are also simple and efficient. During the construction of basic analysis block, the code is disassembled and parsed until encountering a branch instruction or memory accessing instruction, this procedure is implemented in the function *QueryBuildAnalysisBlock* in Fig. 3, the detail is omitted here for simplicity. In addition, instructions related with system calls or privilege transitions are also regarded as the normal end of basic analysis block, because the current analysis aims at the execution in user mode and the EPT fault will occur again when the system call returns from kernel due to the stack memory read, thus keeping the analysis process continuous.

```
Algorithm:  Execution Tracing and Instruction Analysis
Input:      Start address s_ins_addr, end address t_ins_addr
Output:     Analysis results and parsed information of basic analysis blocks.
---------------------------------------------------------------------------
1.  Initialize the analysis environment.
2.  t_addr ← s_ins_addr
3.  while t_addr is legal do
4.          block_info = QueryBuildAnalysisBlock(t_addr)
5.          Check the information included in block_info
6.          if the end address for block_info is t_addr then
7.              break
8.          else if the end instruction for block_info is system call then
9.              break
10.     end if
11.     Switch the context and execute the analysis code cached in block_info.
12.     Obtain the following target of block_info, and assign its address to t_addr.
13. end while
14. Save the analysis results and return.
```

Figure 3: Algorithm of execution tracing and instruction analysis

3.3 Analysis optimizations

Because some programs involve a great deal of memory accesses during the execution such as encryption and compression, which would generate a large number of exceptions by EPT violations in a short time. Then it will consume a lot of processor resources and affect the normal execution of target program. Therefore, to further improve the analysis performance, we should reduce the number of EPT faults while remaining the effectiveness of the interception, which is important on cloud platforms. For this purpose, we continue to propose the following optimizations based on the aforementioned method.

While handling the virtualization exception due to the EPT violation, we also profile the execution of code generating EPT faults. The execution profiler counts the number of faults generated by each code page to discover the one that contains instructions executed very frequently. We call these code pages as hot code, such as the loop code. The hot code may generate a large amount of EPT faults in a very short period, which will lead to the drastic performance reduction of the target program. Therefore, for these pages, we rewrite the original code in the new memory space and redirect the program execution, then the subsequent interception and analysis of the code page will be performed in the inline form to further improve the analysis performance. It is similar to the traditional dynamic instrumentation, we also construct the hybrid executable code that includes the fine-grained analysis code and original instructions of target program, but the difference is that only those instructions involving memory accesses will be intercepted and modified in real time. As shown in Fig. 4, the middle code page is rewritten, when the execution enters the current code segment and triggers exceptions, the subsequent execution will be redirected to the new code, and the interception and analysis will be completed in inline mode, including internal jump instructions, without generating EPT exceptions.

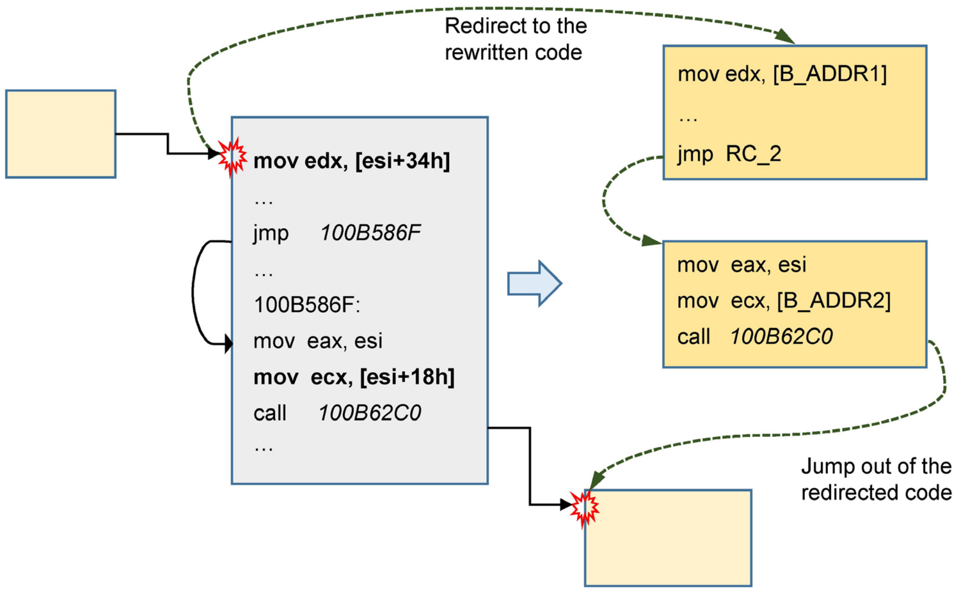


Figure 4: Example of code rewriting and execution

On this basis, various interception functions can be implemented more flexibly and efficiently, such as obtaining function call sequences. The disadvantage is that it will modify part of execution code of target programs and consume more memory address space. However, because the analysis method is in super mode, the kernel address space can be used to construct the redirection code to be executed, thus reducing the interference in the user space.

In addition, in rewriting code, the original virtual address to be accessed needs to be redirected to bypass the existing EPT access control during subsequent execution, thus conducting interception in inline mode. As shown in Fig. 4, both *B_ADDR1* and *B_ADDR2* are redirected addresses. To achieve this, it is necessary to establish a mirror page for each guest physical page allocated by the target program in the guest VM, and map the guest physical address of mirror page to the same host physical address of the mirrored guest physical page. Actually, even the consumption of guest physical addresses is twice as before for the target program, but the addresses in kernel mode can also be used instead. As shown in Fig. 5, when an instruction accesses the virtual address 0×471260, the actual target will be redirected to the virtual address 0×1A2260, and both of them are mapped to the same host physical address 0×595000 through EPT translations. Consequently, the memory access will not generate any EPT violation. In addition, the target program often contains lots of shared code, such as dynamic link library, which shares the same physical pages. In this case, when code rewriting is performed, we could make a copy of the physical page first to avoid the impact on other programs.

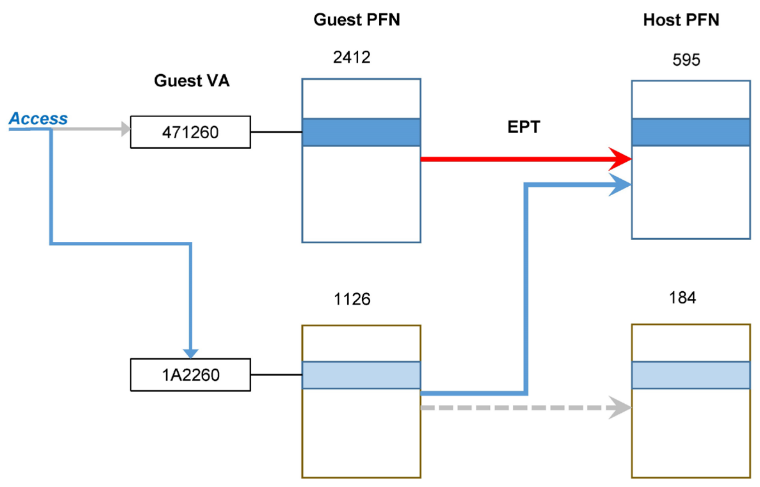


Figure 5: Memory access for bypassing EPT violations

4 Implementation and experiments

The prototype system of this method is implemented on a 32-bit Windows platform. In order to highlight the emphasis and simplify the development, the VM infrastructure code is further developed based on the HyperPlatform released on Github. Therefore, after the hypervisor is loaded and run, the original Windows system will run as a guest virtual machine, similar to Hyper-V technology, thus realizing the virtualization architecture of cloud platforms. The whole analysis code is included in the kernel driver, including VM infrastructure, analysis module and other auxiliary modules. Another tool udis86 is used as the disassembly engine to parse the target instructions.

The main experimental environment of this paper is a common desktop terminal, on which a virtualization environment similar to cloud platforms is built. The hardware configuration is Intel i5-7500 @ 3.40 GHz 4 core CPU, 4 GB memory, 120 GB solid-state system disk and 500 GB data disk (7200 revolutions), and a 32-bit Windows 10 (10240) operating system is installed.

4.1 Performance evaluation and analysis

First, we evaluate the performance impact and check the feasibility by analyzing multiple types of common programs with the base analysis method without optimizations. The list of programs is shown as Fig. 6. Some of the programs are system applications, which use file download functions supported by the built-in programs such as *certutil, regsvr32* and *cscript* to download 1 MB files from a local server. This trick is adopted in many malicious codes at present. Then two third-party network tools *aria2c(1.34)* and *curl(7.61)* are also used to download the same file respectively. In the above test scenarios, https protocol is enabled for *aria2*, and the *gzip* transfer encoding is also tested for *curl*. Similarly, the *qqmusicup*(v16.20) program, a legitimate application component, will also connect to the network and communicate. Additionally, we test the startup and exit process of the built-in graphical program *xpsrchvw*. To make comparisons, we also use Pin to instrument and analyze the programs in above scenarios, trace the instructions with memory accesses and all instructions have been executed, respectively denoted as PIN_M and PIN_T in Fig. 6. Correspondingly, FBAC_I represents the interception execution process while FBAC_A represents the tracing and analyzing all the executed instructions. In the experiments, the execution time of programs under each test scenario is recorded separately. In fact, the time consumption of native execution is less than 1 s for most of the test programs. Repeat each experiment 10 times and average out the results as shown in Fig. 6.

In most cases, the analysis performance of this method is equivalent to that of binary analysis in traditional environments, and on some programs the former is better than the latter. Because for traditional analysis, the analysis code is embedded in the native code and executed directly by the program, which leads to the low runtime performance overhead in the repeated execution. But our method works better for the interception and analysis of new allocated code in the execution of target program. In addition, for network-related applications, better analysis performance can be achieved, which is also the focus of this paper, because privacy disclosure behavior is mostly related to network activities. In the experiments, the data flow propagations between local files and network packets can also be correctly detected for these network programs.

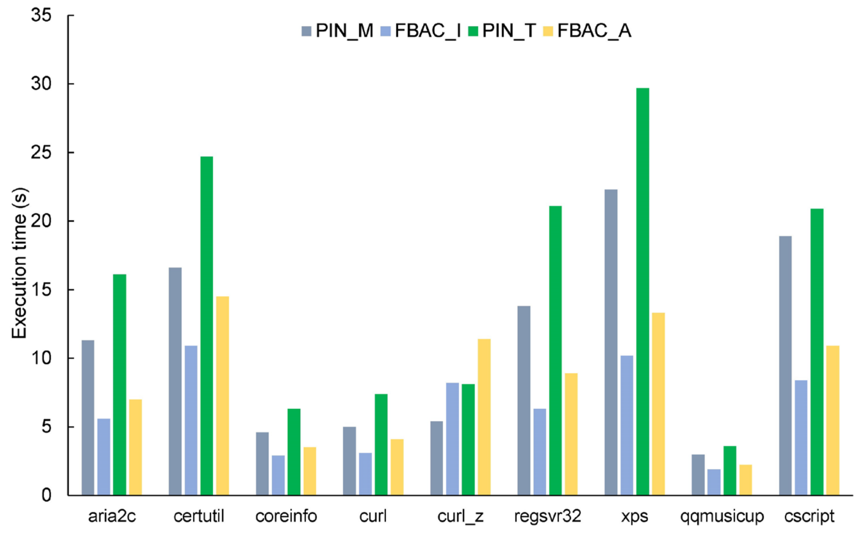


Figure 6: Performance overhead of the base analysis and comparisons

For programs involving plenty of compression and encryption operations, the analysis performance is poor when not optimized, mainly due to the continuous EPT violations generated by a large number of memory accesses which will consume much processor time, as shown in Tab. 1. In fact, the number of faults generated by consecutive instruction addresses account for about 40% of the total in average, which should be optimized. As seen from Tab. 1, lots of violations occur at the same instruction address, so the cached code of basic analysis blocks can help to improve the analysis efficiency. Moreover, most of execution paths to be analyzed between two exceptions are not complicated, avoiding a large number of intermediate execution operations during the analysis, and the slowdown caused by FBAC_A does not change drastically compared with FBAC_I.

Table 1: Other statistics under the base analysis

Program	Number of EPT faults (million)	Number of loaded code pages causing faults	Number of unduplicated fault addresses (k)
aria2c	11	1028	71.6
certutil	18.1	1382	233
coreinfo	3.9	458	28.5
curl	4.9	527	68.6
curl_z	13.1	531	81.7
regsvr32	11.2	1128	116.7
xps	15.8	1525	150.2
qqmusicup	10.5	1045	55.5
cscript	13.4	1517	155.7

In addition, the processor usage increases in the analysis as similar to the traditional method, due to the execution of additional analysis instructions, which depends on the usage of target programs in the native execution as well as the duration of the analysis. Judging from the results, it borders on the complete occupation of a core. Therefore, while controlling fine-grained analysis in practical applications, each target machine can be allocated an extra core for auxiliary analysis. Besides, through the additional experiments we also obtain that the capability of address translation and interrupt response will increase when the hardware is upgraded.

4.2 Evaluation of optimizations

Then we analyze and evaluate the analysis performance adopting the optimization of hot code rewriting. Among the above programs, some with more memory accesses at runtime are chosen, which are *openssl (1.0.2n)*, *7zip (18.05)*, *ffmpeg (4.0.2), pscp (0.70)* and the network application *curl*. The test scenarios stay the same as the above. When the number of EPT faults generated by the instruction reaches the preset threshold, we rewrite the entire code page where it is located. The subsequent interception of the code execution is then performed in the inline form, here we only demonstrate the effect of interception without analyzing all executed instructions. The experimental results are shown as Fig. 7, where FBAC_OPT_C1 indicates that when the hot spot code generates the exception 10,000 times, carry out instrumentation interception, while FBAC_OPT_C2 indicates that the trigger threshold is 1,000 times. As shown in Fig. 7, it is clear that when the interception method for hot spot code segments that are partially repeatedly executed is modified, the interception analysis performance of target programs will be greatly improved, and the test result at FBAC_OPT_C2 is already better than that at PIN_M.

According to Tab. 2 and Fig. 8, the proportion of redirected code pages is not high, and will lead to a sharp drop in the number of EPT exceptions. This affects the performance of different programs differently, mainly due to the density difference of memory access instruction distribution. For example, memory access instructions are mostly located in adjacent positions for 7zip, so the improvement of analysis performance is not obvious at the two thresholds, and its analysis performance is affected by code expansion caused by instrumentation. In the current experiment, the instrumentation code has not been optimized yet. In fact, there are many studies that can be referenced.

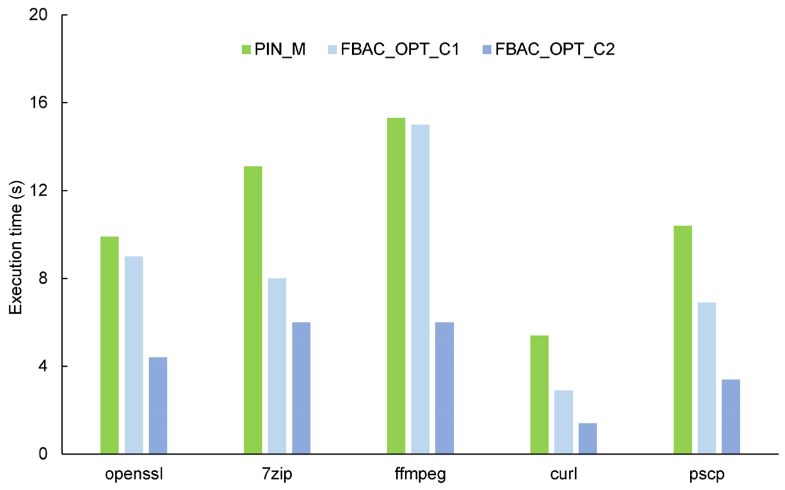


Figure 7: Analysis performance with rewriting of hot code pages

5 Discussion

For the online analysis on cloud platforms, performance is more important than concealment, because excessive overhead may affect the normal execution of target program. Therefore, more system level components are introduced here based on the hypervisor which can improve the analysis performance and reduce the difficulty of acquiring system semantics, it is still stealthy for the user mode analysis.

The proposed method can take over and analyze the target program at any time, and the analysis can also be securely cancelled during the execution. Rewriting hot code involves a small amount of modifications to the target code, but kernel space can be used to reduce the impact. In actual deployment, fine-grained analysis should be turned on and off in due time, and interception scope should be flexibly controlled. In this situation, optimizations can be made according to the characteristics of specific target programs.

The experiments are also confined to the current test environment, but the absolute value of analysis performance can be improved through better hardware. Currently, the analysis of user-level binary programs can also be further expanded to increase support for kernel-mode code. In addition, further experimental analysis is needed for multiple virtual machines environment, other open-source cloud platforms and processor isolation environments.

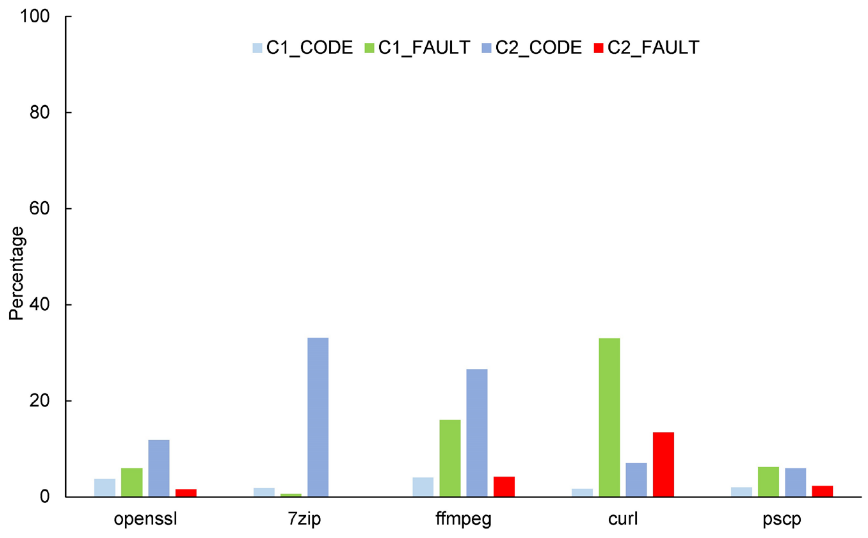


Figure 8: Proportion of hot code and EPT faults to the non-optimized analysis

Table 2: Statistics of hot code rewriting for the test programs

Program	Triggering threshold (k)	Counts of rewritten code pages	Counts of EPT faults (million)
openssl	1	87	3.0
	10	27	11.1
7zip	1	236	0.2
	10	13	4.5
ffmpeg	1	375	6.3
	10	56	24.1
curl	1	53	1.7
	10	12	4.3
pscp	1	64	3.3
	10	23	9.1

6 Conclusion

This paper proposes a new fine-grained monitoring method for privacy leakage detection on cloud platforms. The method can intercept the execution of target programs in real time with the help of multiple underlying components, and perform online dynamic data flow analysis to the possible privacy leakage and program misconfigurations. We also propose the analysis optimizations to further reduce the runtime performance overhead, and perform the efficient analysis according to the analysis scenarios.

Experiments in real environments show that the proposed method and optimizations can be applied for the program monitoring and analysis on cloud platforms, which also have advantages in the analysis of programs with network activities. Further research should be done on optimization methods to improve analysis performance and stability.

Funding Statement: This work is supported by the National Natural Science Foundation of China (General Program, Grant No. 61572253, YZ, http://www.nsfc.gov.cn) and the Innovation Program for Graduate Students of Jiangsu Province, China (Grant No. KYLX16_0384, JP, http://jyt.jiangsu.gov.cn).

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

Bauman, E.; Ayoade, G.; Lin, Z. (2015): A survey on hypervisor-based monitoring: approaches, applications, and evolutions. *ACM Computing Surveys*, vol. 48, no. 1, pp. 1-33.

Chung, C.; Khatkar, P.; Xing, T.; Lee, J.; Huang, D. (2013): NICE: network intrusion detection and countermeasure selection in virtual network systems. *IEEE Transactions on Dependable and Secure Computing*, vol. 10, no. 4, pp. 198-211.

Deng, Z.; Zhang, X.; Xu, D. (2013): SPIDER: stealthy binary program instrumentation and debugging via hardware virtualization. *Annual Computer Security Applications Conference*, pp. 289-298.

Dinaburg, A.; Royal, P.; Sharif, M.; Lee, W. (2008): Ether: malware analysis via hardware virtualization extensions. *ACM Conference on Computer and Communications Security*, pp. 51-62.

Fu, S.; Kim, H.; Prior, R. (2015): FlowBox: anomaly detection using flow analysis in cloud applications. *IEEE Global Communications Conference*, pp. 1-6.

Gupta, S.; Kumar, P. (2015): An immediate system call sequence based approach for detecting malicious program executions in cloud environment. *Wireless Personal Communications*, vol. 81, no. 1, pp. 405-425.

Han, Y.; Hao, Z.; Cui, L.; Wang, C.; Sang, Y. (2016): A hybrid monitoring mechanism in virtualized environments. *IEEE Trustcom/Bigdatase/ISPA*, pp. 1038-1045.

Henderson, A.; Prakash, A.; Yan, L. K.; Hu, X.; Wang, X. et al. (2017): DECAF: a platform-neutral whole-system dynamic binary analysis platform. *IEEE Transactions on Software Engineering*, vol. 43, no. 2, pp. 164-184.

Jee, K.; Kemerlis, V. P.; Keromytis, A. D.; Portokalidis, G. (2013): ShadowReplica: efficient parallelization of dynamic data flow tracking. *ACM SIGSAC Conference on Computer and Communications Security*, pp. 235-246.

Kemerlis, V. P.; Portokalidis, G.; Jee, K.; Keromytis, A. D. (2012): Libdft: practical dynamic data flow tracking for commodity systems. *ACM Sigplan Notices*, vol. 47, no. 7, pp. 121-132.

Korkin, I.; Tanda, S. (2017): Detect kernel-mode rootkits via real time logging & controlling memory access. *Annual Conference on Digital Forensics, Security and Law*.

Lee, J. K.; Moon, S. Y.; Park, J. H. (2017): CloudRPS: a cloud analysis based enhanced ransomware prevention system. *Journal of Supercomputing*, vol. 73, no. 7, pp. 3065-3084.

Lengyel, T. K.; Maresca, S.; Payne, B. D.; Webster, G. D.; Vogl, S. et al. (2014): Scalability, fidelity and stealth in the DRAKVUF dynamic malware analysis system.

Annual Computer Security Applications Conference, pp. 386-395.

Liu, J. K.; Liang, K.; Susilo, W.; Liu, J.; Xiang, Y. (2016): Two-factor data security protection mechanism for cloud storage system. *IEEE Transactions on Computers*, vol. 65, no. 6, pp. 1992-2004.

Liu, Z.; Xia, J. (2019): A cross-tenant RBAC model for collaborative cloud services. *Computers, Materials & Continua*, vol. 60, no. 1, pp. 395-408.

Luk, C. K.; Cohn, R.; Muth, R. (2005): Pin: building customized program analysis tools with dynamic instrumentation. *ACM Sigplan Notices*, vol. 40, no. 6, pp. 190-200.

Ming, J.; Wu, D.; Xiao, G.; Wang, J.; Liu, P. (2015): TaintPipe: pipelined symbolic taint analysis. *24th USENIX Security Symposium*, pp. 65-80.

Mishraa, P.; Pillia, E. S.; Varadharajanb, V.; Tupakula, U. (2017): Intrusion detection techniques in cloud environment: a survey. *Journal of Network and Computer Applications*, vol. 77, pp. 18-47.

Pan, J.; Zhuang, Y.; Sun, B. (2020): BAHK: flexible automated binary analysis method with the assistance of hardware and system kernel. *Security and Communication Networks*, https://doi.org/10.1155/2020/8702017.

Patel, A.; Taghavi, M.; Bakhtiyari, K.; Ju'nior, J. C. (2013): An intrusion detection and prevention system in cloud computing: a systematic review. *Journal of Network and Computer Applications*, vol. 36, no. 1, pp. 25-41.

Rocha, F.; Gross, T.; Moorsel, A. (2013): Defense-in-depth against malicious insiders in the cloud. *IEEE International Conference on Cloud Engineering*, pp. 88-97.

Schwartz, E. J.; Avgerinos, T.; Brumley, D. (2010): All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). *IEEE Symposium on Security and Privacy*, pp. 317-331.

Vogl, S.; Eckert, C. (2012): Using hardware performance events for instruction-level monitoring on the x86 architecture. *Proceedings of the 2012 European Workshop on System Security*.

Willems, C.; Hund, R.; Fobian, A.; Felsch, D.; Holz, T. (2012): Down to the bare metal: using processor features for binary analysis. *Annual Computer Security Applications Conference*, pp. 189-198.

Zhang, T.; Lee, R. B. (2018): Design, implementation and verification of cloud architecture for monitoring a virtual machine's security health. *IEEE Transactions on Computers*, vol. 67, pp. 799-815.