EMITTER International Journal of Engineering Technology
ISSN: 2443-1168, Vol. 8, No. 1, June 2020, pp. 49~66
DOI : 10.24003/emitter.v8i1.468

Towards a Resilient Server with an External VMI in the Virtualization Environment

Agus Priyo Utomo, Idris Winarno, Iwan Syarif

Departement of Information and Computer Engineering,
Politeknik Elektronika Negeri Surabaya
Jl. Raya ITS Sukolilo Surabaya 60111, Indonesia
E-mail: agusp@poliwangi.ac.id, {idris, iwanarif}@pens.ac.id

Received December 31, 2019; Revised May 3, 2020; Accepted May 25, 2020

Abstract

Currently, cloud computing technology is implemented by many industries in the world. This technology is very promising due to many companies only need to provide relatively smaller capital for their IT infrastructure. Virtualization is the core of cloud computing technology. Virtualization allows one physical machine to runs multiple operating systems. As a result, they do not need a lot of physical infrastructures (servers). However, the existence of virtualization could not guarantee that system failures in the guest operating system can be avoided. In this paper, we discuss the monitoring of hangs in the guest operating system in a virtualized environment without installing a monitoring agent in the guest operating system. There are a number of forensic applications that are useful for analyzing memory, CPU, and I/O, and one of it is called as LibVMI. Drakvuf, black-box binary analysis system, utilizes LibVMI to secure the guest OS. We use the LibVMI library through Drakvuf plugins to monitor processes running on the guest operating system. Therefore, we create a new plugin to Drakvuf to detect Hangs on the guest operating system running on the Xen Hypervisor. The experiment reveals that our application is able to monitor the guest operating system in real-time. However, Extended Page Table (EPT) violations occur during the monitoring process. Consequently, we need to activate the altp2m feature on Xen Hypervisor by minimizing EPT violations.

Keywords: Virtualization, Virtual Machines Introspection, out-VMI, Hang Detection, Cloud Computing.

1. INTRODUCTION

Information technology is growing very fast so that new technologies are rapidly replacing current projections. One of them as cloud computing technology, which offers easy access, efficiency, and scalability of network resources. As a result, cloud computing has made many industries migrating

their IT infrastructure to it. Cloud computing is also able to change the lifestyles of the world community, many of which they associate with technology (e.g., IoT), and even they also visualize it in the form of mobile applications. Simply by using a computer or mobile phone, they can do many things such as reading news, teaching and learning activities, buying and selling transactions, turning off remote lights, and even they are able to control the growth of crops in the agricultural sector (e.g. agrotechnology). In 2021, it is estimated that the data stored in the data center will reach 1.3 ZB, up 4.6 times from the previous 286 EB in 2016, this is a finding of the Cisco Global Mobile Data Traffic Forecast (2016 to 2021) [1]. With more and more people and objects connected to each other, we need a connecting medium called an information system. There are some components in information system and one of the essential parts called computer server. Therefore, the availability of the servers is very important to avoid companies from revenue loss when access to data resources and business applications is hampered [2]. Building a server can be done with a traditional model and can also use a virtualization model. Traditional server architecture is different from virtualization server architecture as shown in Figure 1.

Traditional servers are also known as bare-metal servers. Each traditional server consists of physical devices in the form of memory, processor, I/O, network connection, hard drive, and operating system (OS) to run programs and applications. The OS works directly on physical devices that depend entirely on the availability of supporting hardware resources. Unlike virtualization servers, this operates in a "multi-tenant" environment, which means that multiple virtual machines (VMs) runs on the same physical hardware. In this case, the computing resources of the physical server are virtualized and shared among all the VMs that run it. The virtualization server architecture is a bit more complicated than a physical server. This requires a hypervisor, which is used to create and manage VMs, which have their own virtual computing resources. Furthermore, we can create several guest OS and application servers on top of virtual hardware.

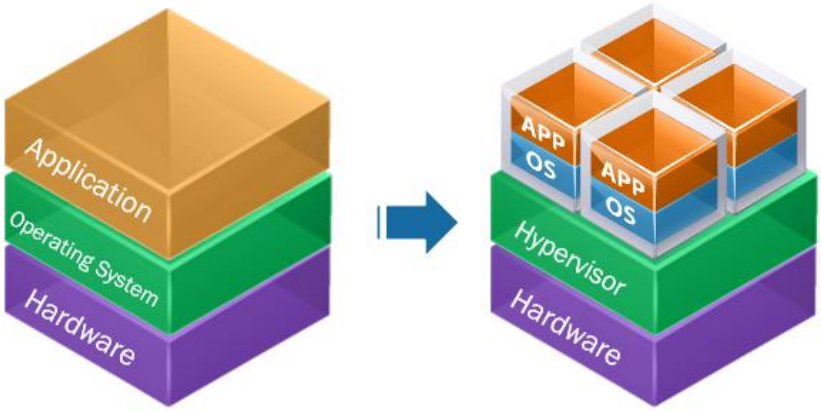

Figure 1. Traditional Architecture vs Virtualization Architecture

The hypervisor is divided into two types, namely Bare Metal (Native) and Hosted. Native hypervisor type runs concurrently with server hardware, so it has direct access rights to the hardware without having to pass another operating system. That is, it does not require another operating system to run this type of Hypervisor (e.g., VMware ESXi, Xen Server). Hosted hypervisor type acts as software that is useful for running and managing virtual machines, so to get access to the hardware must pass through the operating system (e.g., Proxmox, VirtualBox, Xen Hypervisor, KVM Hypervisor). Thus, server virtualization makes it possible to run multiple OS and applications based on shared physical hardware, which makes it a cost-effective choice for building servers rather than traditional servers [21].

The impact is that many traditional server users switch to virtualization servers, this is due to the many tangible benefits offered by the virtualization model including ease of use and scalability [3]. The Cisco Global Cloud Index estimates [1] that, in 2021, the ongoing transition from workloads and calculation of traditional data center instances to cloud data centers is 94%, as shown in Figure 2.

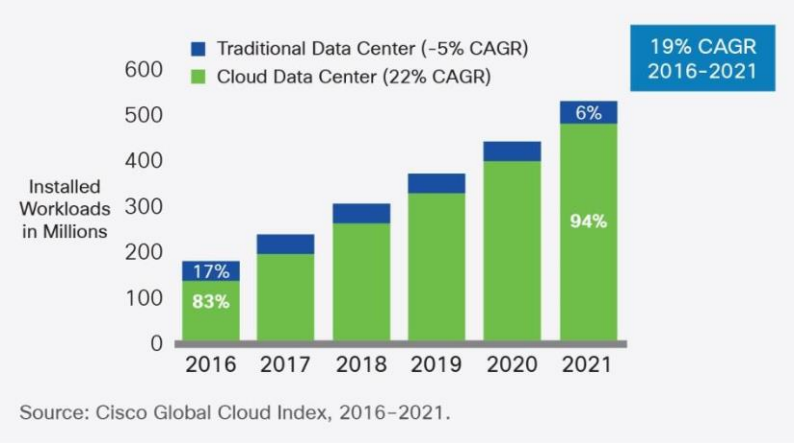

Figure 2. Workload and compute instance distribution [1]

However, shifting the infrastructure from traditional servers to virtualization does not guarantee that the server will avoid potential failures or other new threats. Consequently, the more new features or services will deliver, more potential failures that occur such as Hang (system failure), Denial of Service (DoS), and Malware (Hidden Rootkit) [4][5]. Virtual Machine Introspection (VMI) is a solution that can be used to solve the problems that occur in virtual machines and create a resilient computing ecosystem.

2. RELATED WORKS

The VMI concept was first introduced by Garfinkel and Rosenblum in 2003 [6]. Then according to Zhao et al. [7] this technique is distinguished according to how it works namely in-VMI is a monitoring technique from within the guest OS, the detection application acts as an agent that can be run

manually or as a service on the operating system, whose task is to monitor the system that is running on the guest OS. And out-VMI is a guest OS monitoring technique without installing an application (agent) inside the guest OS. The more guest OS running on the host OS, the more efficient this method is. Many different VMI techniques have been developed in the last few years, starting with the introduction of the semantic gap problem. It can be concluded that in the case of developing computational security, the use of VMI has received a serious response [6][8].

In the operating system, there is a paging mechanism, one of which is Second Level Address Translation (SLAT). SLAT serves to bridge the virtual address of the guest OS with physical memory, and SLAT is also referred to as an additional layer that mappings the address from the guest OS to physical memory. This technology was initiated by Intel since the Core i3 processor until now under the codename Nehalem. Then AMD introduced the same technology since the third-generation Opteron under the name Rapid Virtualization Indexing (RVI) technology. At present, both Intel Virtualization Technology (VT-x) and AMD Secure Virtual Machine (SVM) have utilized SLAT technology for virtualization needs, which later on Intel processors known as Extended Page Tables (EPT) which are used for Memory Management Units (MMU) [9].

In the virtualization mechanism, EPT mapping Guest Virtual Address (GVA) to be used on the Guest Physical Address (GPA) in the main memory. Each guest OS has a one-page table that is managed by a Virtual Machine Monitor (VMM) to produce physical addresses and other page tables, and this aims to translate the guest OS physical address to the host OS physical address. This process aims to ensure that each memory access operation on the MMU EPT automatically gets the host OS physical address from the results of the mapping performed by VMM. For introspection in a virtual machine, there are several important steps that must be taken. The first step to analyzing something is to collect data/evidence. Data or evidence collection starts with something that is easily changed (e.g., Random Access Memory (RAM)). Data that is in the computer RAM will be lost when the computer has restarted or shutdown.

Therefore, it is important to really pay attention when the server computer is indicated a security incident is not to shutdown or restart the server, which may still hold valuable evidence. RAM itself is just a series of zeros and ones, with no semantic context at all. However, computer programs need to adjust this RAM so they can store meaningful data. For example, in the C programming language, a programmer can define a struct that determines how variables are placed in RAM. We take advantage of the unique Rekall approach to memory analysis, one of which is utilizing the appropriate debugging information provided by the operating system to precisely find significant kernel data structures. Currently, Rekall also provides cross-platform solutions, including Windows, Mac OSX, and Linux.

Some research on the introspection of virtual machines has been done by several researchers before. Pham et al. [4] introduced reliability and security (RnS) in the HyperTap framework to support monitoring in virtualization environments, especially kernel-based virtual machines (KVM). Researchers introduce three examples of monitoring, one of which is Guest OS Hang Detection (GOSHD). The researcher assumes that when the CR3 register is disappear in a span fewer than 10 minutes and appears again then the guest OS can be said to be in a partial hang condition. Winarno et al. [10] evaluate the performance of in-VMI virtualization servers by developing agents as an in-VMI application and referred to as the Self-Repair Network (SRN) manager. Three failure scenarios are tested, one of which is hang. To detect hang, they use the heartbeat probes method. Lengyel et al. [11] build a malware analysis system that dynamically runs on virtualization (Drakvuf). The Drakvuf monitoring system is able to monitor rootkits in kernel-mode on the guest OS. Drakvuf works on the host OS to monitor guest OS by utilizing memory access using LibVMI, functioning as a malware detection system at the guest OS kernel level.

3. ORIGINALITY

In this work, we discuss the Hang detection system (system failure) on the guest OS. We focus on the out-VMI method, we focus on this because in the cloud business world it is impossible for providers to interfere to a guest OS that has been leased without having access rights to log in. However, the out-VMI method is able to provide solutions to problems of access rights, meaning that out-VMI can monitor many guest OSes running on Virtual Machines without interference the guest OS. In contrast to existing research, they developed a hang detection system on the Kernel-Based Virtual Machine (KVM) [4], and a malware detection system on Xen Hypervisor-based virtualization [11]. In our previous study [12], we developed a hang detection system on the Xen Hypervisor, and have successfully read the Control Register (CR3) of each process running on the guest OS as an indicator that the guest OS is running normally. The CR3 register is part of a memory register that can translate linear addresses into physical addresses when virtual addressing is activated. We use the CR3 register to find out all the Process ID (PID) that are running at the Operating System level in the guest OS. Furthermore, we continue our work to detect the partial hangs of the applications that run under the Hypervisor.

Table 1. Comparison of differences with existing research

Existing Study	Engine	CR3	Malware	Partial Hang	Full Hang	eVMI	iVMI	Framework
Pham et al, 2014 [4]	KVM	√	√	√	√	√		HyperTap
Lengyel et al, 2014[11]	XEN	√	√			√		LibVMI
Winarno et al, 2018 [10]	XEN, KVM		√		√		√	SRN
Utomo et al, 2020	XEN	√		√		√		LibVMI

4. SYSTEM DESIGN

This study aims to detect a partial hang on the guest OS running on a virtual machine based on Xen Hypervisor. We propose the use of the out-VMI method. In order, to get optimal results, it requires several steps starting from getting debug information for the guest OS kernel to the process of detecting a partial Hang on the guest OS.

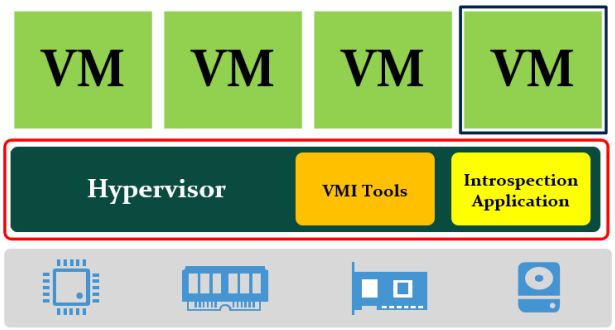

Figure 3. Out-VMI Architecture

When the guest OS in a virtualized environment experiences problems in the middle of the process, the guest OS condition will never be known by the system administrator. Failures that occur in the guest OS can be caused by failures in the hardware (e.g. network connection problem) or logic failures (e.g. software bugs). Therefore, several guest OS monitoring applications appear on the hypervisor. However, it requires the installation of additional applications to the guest OS to send information about the failures that occur on the guest OS. When using the out-VMI method to monitor guest OS conditions, it is not necessary to install additional applications since the monitoring process is done from the outside of guest OS as shown in Figure 3.

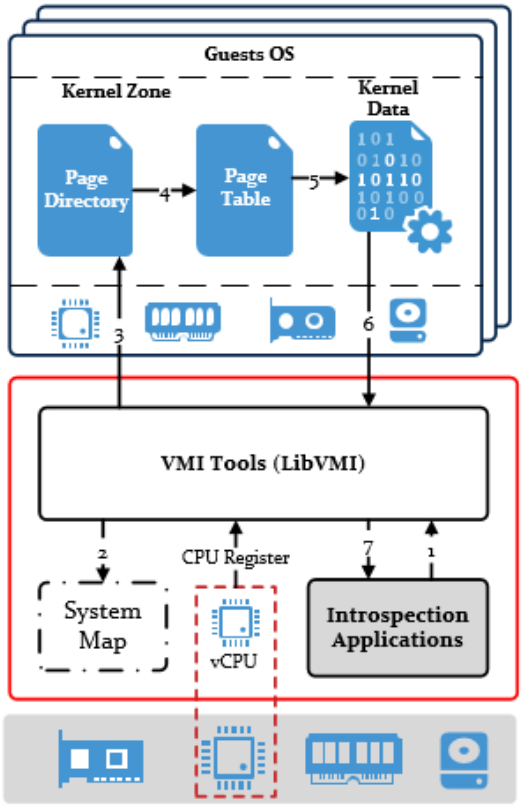

Figure 4. System Design Hang Detection

We developed an external virtual machine introspection (VMI) application that runs on the host OS to detect system failures (e.g., Hang) on the guest OS. The external VMI detects the system failures by reading the CR3 register that indicates the process is running or not at the memory level. That is, the kernel scheduler does not forward PIDs that are running at the OS level to the next process that is processed at the memory level as shown in Figure 4.

4.1 System Map

In order to run LibVMI and read processes in memory, it requires a kernel symbol from each guest OS to monitor, and this is called a System Map. On a guest OS that uses Windows, a system map can be obtained by translating the kernel symbol (e.g., Ntoskrnl.exe). Furthermore, to get debugging information from the kernel, we need to provide a Globally Unique Identifier (GUID) and Database Program (PDB). In Windows guest OS, we could get the kernel to debug the information (e.g., GUID, PDB code) via the vmi-win-guid tool, which is part of LibVMI. Further, creates a debugging information request addressed to the Microsoft site, based on the GUID and PDB code that has been obtained using the Rekall application [18]. When the debugging information has been downloaded successfully, we do the conversion into a JSON file format, as shown in Figure 5.

However, on a guest OS based on Linux, the process to get the System Map occurs a bit difference. Debugging information is obtained directly from the guest OS itself by using the Rekall application. Afterward, the debug information file obtained from extracting the guest OS kernel and transferred to the host OS using a data transfer application (e.g., Secure Copy Protocol (SCP)). Furthermore, on the host OS, we convert the debug information that has been obtained from the guest OS into the JSON format using the Rekall application, as shown in Figure 6. The resulting conversion file is also referred to as the Rekall Profile.

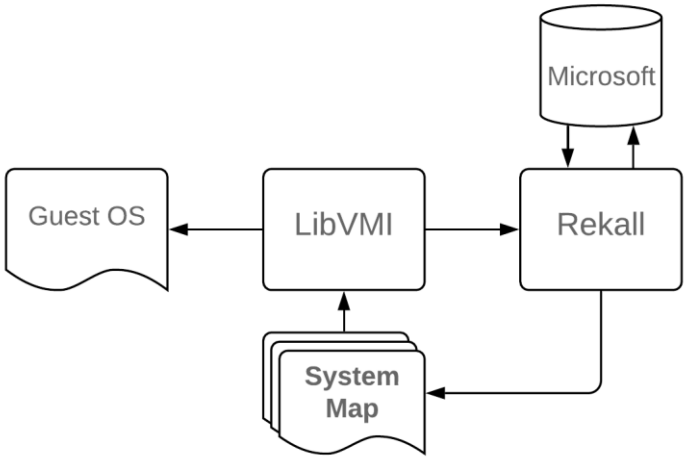

Figure 5. The process of getting a System map on Windows

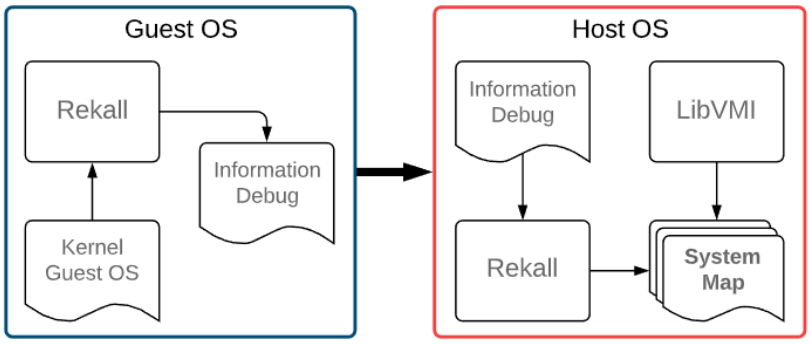

Figure 6. The process of getting a System Map on Linux

4.2 Monitoring

The virtual machine introspection library, known as LibVMI [11], is an additional project of XenAccess. LibVMI is integrated to a memory forensic in order to provide a simpler software so that it makes easier to support virtual machine introspection. This accesses memory using physical or virtual addresses and kernel symbols, kernel symbols obtained from the guest OS kernel data that are translated into files in the JSON format, as shown in Figure 7a. Therefore, LibVMI can provide access to physical memory for each Windows and Linux operating system by using the guest OS kernel symbol, as shown in Figure 7b. The diagram of the memory access process on this

virtual machine is shown in Figure 7. In addition to access the memory, LibVMI also supports access to memory events. LibVMI able to provides a notification of the list of processes that are being run, written, or read when the memory event on the virtual machine is accessed. Memory events require Hypervisor support, which is currently available on XEN and KVM.

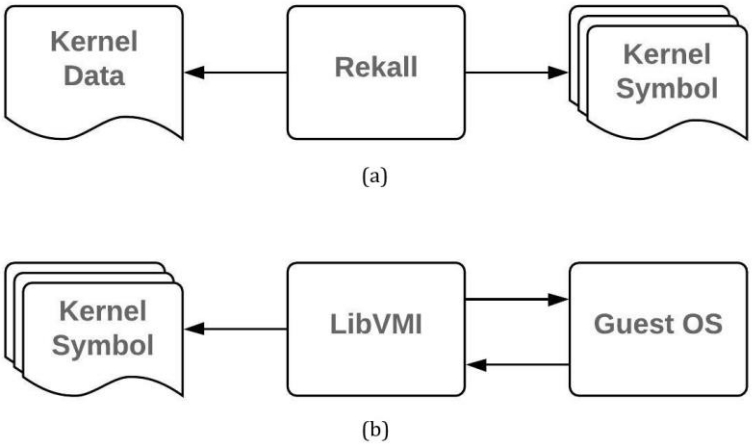

Figure 7. Kernel translation process diagram on VMI

As shown in Figures 5 and 6, to read the CR3 register, it is necessary to translate the kernel symbol into a virtual address. LibVMI requires a System Map of the Guest OS that is associated with the Host OS kernel in order to complete the process. If the System Map file is unavailable, the guest OS monitoring process will fail [13]. The System Map is basically a symbol table and the guest OS kernel address, as shown in Figure 4. The process of scanning a System Map will be carried out continuously by LibVMI until the intended symbol is found. We use LibVMI as a solution that can provide access to the guest OS virtual memory and to read the guest OS virtual address page table, which is basically generated by the guest OS kernel. In every running process, the location of the page directory can change. Therefore, LibVMI scans the kernel task list to find the process page directory based on the given process ID. After finding a match, the page directory will get the process struct, which is in the guest OS virtual memory.

Each PID at the guest OS has a CR3 register at the memory level. This is generated by the kernel scheduler when sending process signals to be executed in memory. We utilize the signal to simulate a partial hang, as shown in algorithm 1.

4.3 Hang Simulation

In the Linux operating system, the kernel has several levels of page tables. The top-level is the global page directory, and each process has a directory page. Thus each process can have a unique table mapping. Because the kernel manages the scheduling process, it can track changes in the page table, and update the CPU status needed to switch to the new process page

table when changing tasks. In the x86 architecture, page directories and page tables together provide a mapping between virtual addresses (memory addresses used by applications) and physical addresses (actual locations in physical memory hardware). When the user runs the process, the CR3 CPU register stores the physical address of the process page table, because each process has a page table, each process will also have a unique CR3 value from every other scheduled process.

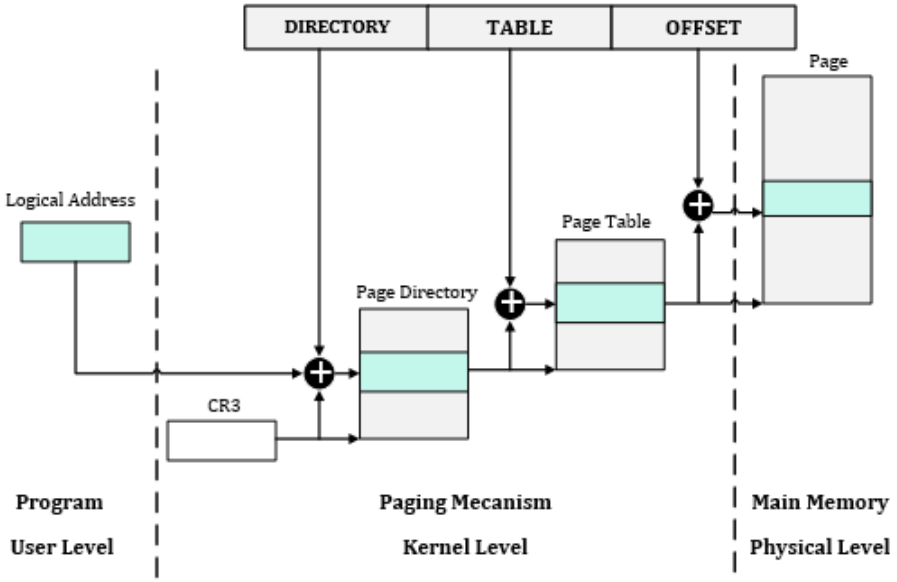

Figure 8. Paging in x86 architecture processor

In a healthy guest OS, every normal process always produce a CR3 register at the memory level. This means that the process IDs running at the OS level will be written continuously at the CR3 register by the process switch. Furthermore, each process in the guest OS has a unique page table. Then each running process will be scheduled by the switch to be processed at the memory level so that every normal process at the OS level will always generate a CR3 code at the memory level. Therefore, when the CR3 register is not written for a period of time (e.g., 10 minutes) by a particular process then, the guest OS can be considered to be in a hang condition. We use the LibVMI library to make it easy to read the CR3 registers for each guest OS in memory. Several studies have conducted approaches to hang detection on virtual machines (e.g., fault injection) [4][14].

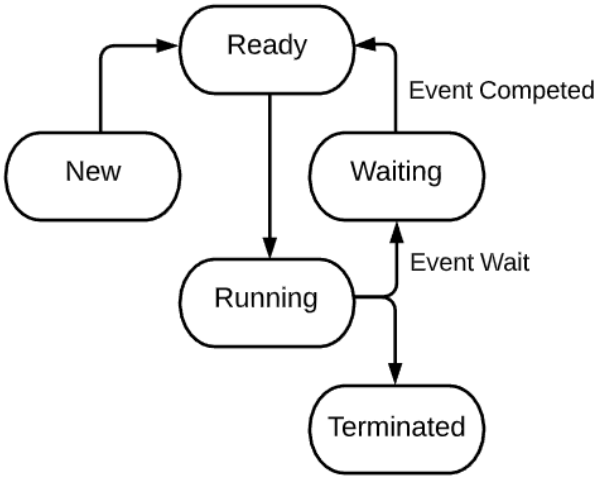

Figure 9. Process cycle in the operating system

In general, Unix-like systems do not have a mechanism to delay the process and then continue. However, this can be manipulated by using the kill application with the SIGSTOP signal [15]. SIGSTOP signal is used to hold or delay the process so that the process switch does not schedule a signal to the page directory. As shown in Figure 9, when this SIGSTOP signal is sent to the process switch, the kernel will actually delay the process from the application (e.g., Apache2). The application process will remain in a waiting state until a SIGCONT signal is sent to be able to run the application process, although the process ID at the OS remains in the running status. However, at the scheduler, this process is suspended. Therefore, we use SIGSTOP to hold the Apache2 process signal from being sent to the process switch. As a result, the CR3 code of the Apache2 services is not written on the CR3 register.

5. EXPERIMENT AND ANALISYS

In this section, we will describe the results of a system failure detection (hang) experiment that occurred on a guest OS, the impact that occurs when the CR3 register reads at the memory level, and the solution to the impact that occurs. To support this experiment, our computer uses Debian GNU Linux 9.6. We compile Xen Hypervisor v4.11 as Hypervisor for its virtualization (Host OS). It runs on physical computer as a server with technical specification as follows:

- Intel (R) Xeon (R) CPU specifications E5-2630 v4 @ 2.20GHz
- 16GiB RAM
- 1TiB HDD

For guest OS, we use Ubuntu 16.04 and allocate 2 vCPU, 1GiB RAM, and 20GiB virtual hard disk.

5.1 Partial Hang Detection

As explained in section 4, especially on subsection 4.3, every process running at the OS level has a process ID. That is, all process IDs running on

the OS print the CR3 register code continuously at the memory level. In this experiment, we ran a web server application service (Apache2) and ran a benchmark application (e.g., UnixBench, stress-ng). The benchmark application is used to force the guest OS to use all of its vCPU resources. Meanwhile, we monitor memory levels by reading guest OS system calls via the host OS. We focus our attention on the Apache2 process (PID) running on the guest OS, and CR3 attached to the service. As a result, monitoring the processes running on the guest OS will be much easier since we do not need any application to the Guest OS. Before performing a partial hang test, we first monitor the memory level to ensure that the CR3 running the Apache2 process has been monitored from the host OS as shown in Figure 10.

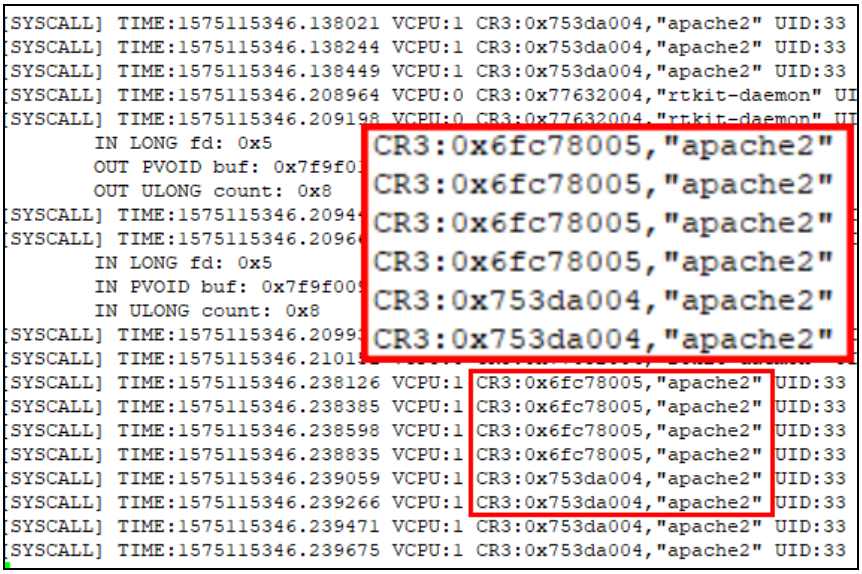

Figure 10. System monitoring produce CR3 for Apache2

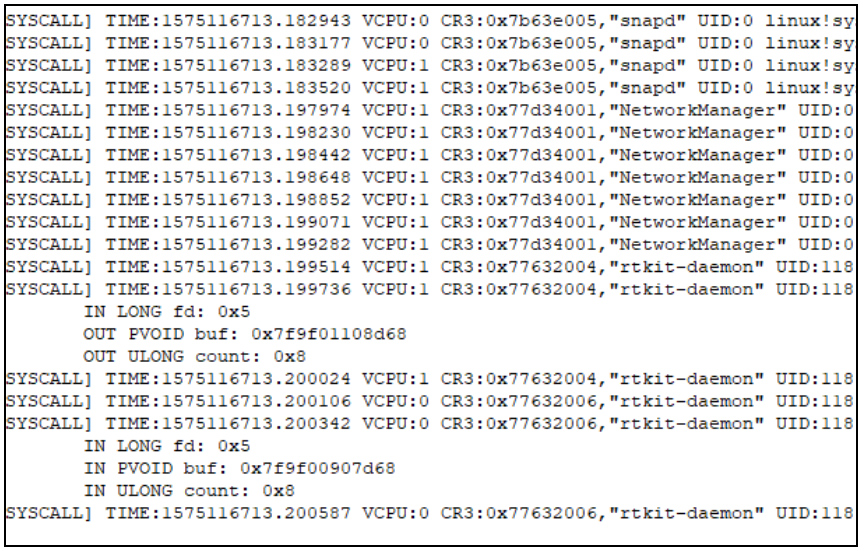

Figure 11. System monitoring not produce CR3 for Apache2

However, when we instructed that the Apache2 signal enters the waiting room with SIGSTOP, the CR3 monitoring system from the Apache2 process is no longer printed at the memory level as shown in Figure 11. According to [4], when CR3 of the application or service is not printed on a monitoring system at the memory level for a specified time, it means that the guest OS is in partially hang condition.

Algorithm 1 Pseudo-code implementation partial hang detection

```
array proclist[1..n]
partialTime = 299
while true do
        result = getSignal()
        if ( result ) then
                procName = result.procName
                time = result.time
                proclist[procName]=time
        end if
        for key in proclist
                haveData = true
                result = getProcList()
                for item in result do
                        if ( item == key ) then
                                break;
                        else
                                haveData = false
                        end if
                end for
                if ( haveData == false ) then
                        remove proclist[key]
                else
                        timeDelay = Time.Now() - proclist[procName]
                        if ( timeDelay > partialTime ) then
                                Print "Partial hang..."
                        end if
                end if
        end for
end while
```

In implementing a partial hang detection system, we use an algorithm as shown in algorithm 1. Our system reads and checks the processed signal from the guest OS based on the CR3 register that appears, then establishes it as a list of processes. After completing registering the process that sent the signal, we then retrieve a list of processes from the guest OS task manager. We do a check between the CR3 registers generated and the list of processes that have been collected previously, if the name of the process is available in the process list it will be ignored. However, if the list of processes does not exist, we delete the list of processes that do not exist in the guest OS process. We also check the length of time the process register does not have a CR3

register. We set a maximum limit of 10 minutes for each process that does not have a CR3 register, as an indicator determining the partial hang.

5.2 EPT Violation

After we successfully indicated there was a partial hang on the Apache2 application. A new problem arises when we read the process in memory. The workload of the guest vCPU OS moves partially to the host vCPU OS. This situation known as EPT violation.

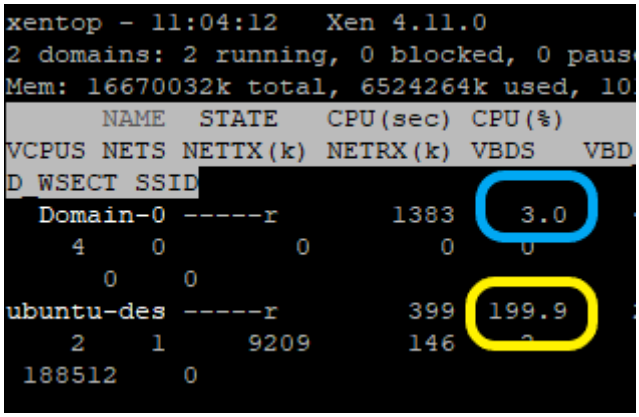

Figure 12. Monitoring the guest OS vCPU usage

EPT violation is detected when we monitor vCPU workloads using monitoring tools (e.g,, Xentop) [20]. The monitoring process is started before reading the memory until resuming the SIGSTOP process performed on Apache2. As shown in Figure 12, the percentage of vCPU usage in the guest OS before reading memory with the guest OS status running the benchmark application reaches 199.9%; this is shown in the yellow rectangle. Moreover, the blue rectangle shows the percentage of use of host OS vCPU with the load running dom0 and one domU (guest OS) by 3.0%.

However, when we monitored memory, the percentage of vCPU usage on the guest OS dropped dramatically to 0.5%. It can be seen in the green rectangle in Figure 13; the use of host OS vCPU has increased to 105.3%. There is a shift of workload from the guest OS to the host OS, and this is called EPT Violation [4].

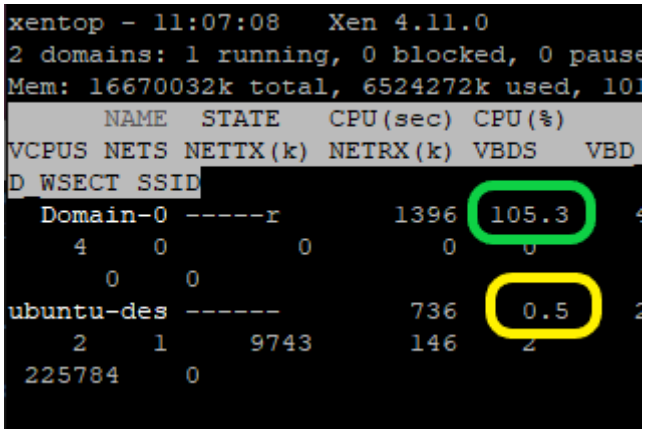

Figure 13. EPT Violation

5.3 Xen Alternate p2m

Altp2m is a Xen subsystem and stands for alternative guest physical memory to machine physical [19]. The term p2m refers to the second level address translation table, which translates the physical guest address (p) into a physical host or in the Xen jargon machine address (m). Usually, there is one p2m table per guest domain that is managed by Xen. This has been changed with an alternative p2m table. On x86 or more precisely on modern Intel architecture, p2m tables can be considered as EPT tables represented by Extended Page Table Pointers (EPTP) in data structures that are determined by the Virtual Machine Control Structure (VMCS) hardware. The altp2m system can define multiple page tables for each guest OS that allows synchronization between vCPUs. Therefore, it can minimize EPT violations in the Hypervisor. Since the Haswell CPU generation, Intel's VMCS has been able to maintain 512 EPTPs. We try to use altp2m on Xen 4.12.1 version as shown in Figure 14.

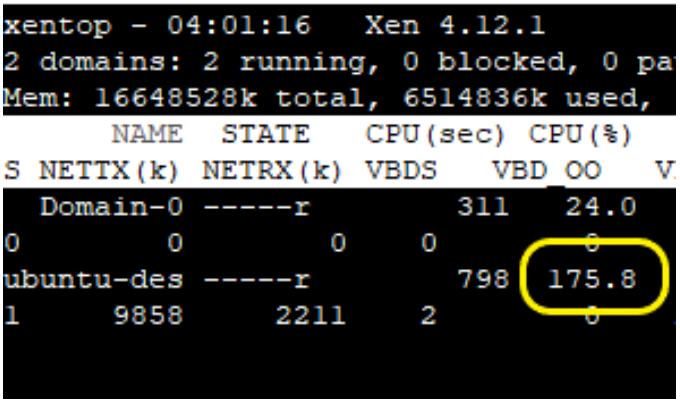

Figure 14. Prosentase vCPU with alt2pm

As a result, when monitoring EPT violations can be minimized. Previously the percentage of vCPU Host OS reached 105.3% as shown in Figure 12. However, since using altp2m it has become 24.0% with the same testing load as shown in Table 2.

Table 2. System Experiment result

Domain	Percentage vCPU		
	Before monitoring	During monitoring	With altp2m
Host OS	3.0%	105.3%	24.0%
Guest OS	199.9%	0.5%	175.8%

6. CONCLUSION

This paper presents monitoring and detection of partial hangs in a hypervisor-based virtualization environment (e.g. Xen) with a virtual machine introspection (VMI) method that is more specific to out-VMI. Our proposed system is able to monitor multiple guest OSes without installing additional applications (agentless). We conducted a simulation for the Hang experiment using a SIGSTOP signal for 10 minutes. As a result, the SIGSTOP signal gives the scheduler instructions to pause the Apache2 application process so as not to forward it to the memory level, and the Apache2 PID is still running at the operating system level. We conclude that our experiment was able to detect a partial Hang on the guest OS. However, when LibVMI monitors the CR3 register, a process shift from the guest OS to the host OS occurs, known as a violation of the EPT. Therefore, we are trying to update to a newer version of Xen. Alternate p2m (altp2m) is able to minimize the occurrence of EPT violations when monitoring memory. We will try to improve our work by adding a self-action model to respond to failures that occur on the guest OS (e.g. Hang) in the future. One of the self-action models is called the Self-Repair Network model [17]. Moreover, the failures not only addresses hang but also other failures such as Malware and Denial of Service (DoS).

Acknowledgements

The author would like to thank the Ministry of Research, Technology and Higher Education, Directorate General of Science, Technology, and Higher Education Resources for their support through the 2018 PTNB Affirmation Scholarship Program.

REFERENCES

[1] Cisco Systems, "**Cisco Global Cloud Index: Forecast and Methodology, 2016–2021**," *Cisco System. Inc*, p. 46, 2018.

[2] A. T. Mizrak, P. Saxena, **VMware vCenter Server High Availability Performance and Best Practices**. *VMware Inc*, 2016.

[3] O. Nagesh, T. Kumar, and V. Venkateswararao, "**A survey on security aspects of server virtualization in cloud computing,**" *Int. J. Electr. Comput. Eng.*, vol. 7, no. 3, pp. 1326–1336, 2017.

[4] C. Pham, Z. Estrada, P. Cao, Z. Kalbarczyk, and R. K. Iyer, **Reliability and security monitoring of virtual machines using hardware architectural invariants**, *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN,

vol. 2014, pp. 13–24, 2014.
[5] T. Y. Win, H. Tianfield, Q. Mair, T. Al Said, and O. F. Rana, **Virtual machine introspection**, *ACM Int. Conf. Proceeding Ser.*, vol. 2014-Septe, pp. 405–410, 2014.
[6] T. Garfinkel and M. Rosenblum, **A Virtual Machine Introspection Based Architecture for Intrusion Detection**, *Proceedings of Network and Distributed Systems Security Symposium,* vol. 1, pp. 253–285, 2003.
[7] S. Zhao, X. Ding, W. Xu, and D. Gu, **Seeing Through The Same Lens: Introspecting Guest Address Space At Native Speed**, *Proceedings of the 26th USENIX Security Symposium, pp. 799-813,* 2017.
[8] L. Jia, M. Zhu, and B. Tu, **T-VMI: Trusted Virtual Machine Introspection in Cloud Environments**, *Proceeding of 17th IEEE/ACM Int. Symp. Clust. Cloud Grid Comput. CCGRID 2017*, pp. 478–487, 2017.
[9] VMware, **Performance Evaluation of Intel EPT Hardware Assist**, Management, vol. 136362, pp. 1–14, 2009.
[10] I. Winarno, Y. Ishida, and T. Okamoto, **A Performance Evaluation of Resilient Server with a Self-Repair Network Model**, *Mobile Networks and Applications*, pp. 1095-1103, 2018.
[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**, *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC 2014, pp. 386–395, 2014.
[12] A. P. Utomo, I. Winarno, I. Syarif, **Detecting Hang on the Virtual Machine using LibVMI**, *2019 International Electronics Symposium (IES)*, Surabaya, pp. 618–621, 2019.
[13] M. A. A. Kumara and C. D. Jaidhar, **Execution time measurement of virtual machine volatile artifacts analyzers**, *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS*, vol. 2016-January, pp. 314–319, 2016.
[14] B. Teabe, V. Nitu, A. Tchana, and D. Hagimont, **The lock holder and the lock waiter pre-emption problems: Nip them in the bud using informed spinlocks (I-Spinlock)**, *Proceeding of 12th Eur. Conf. Comput. Syst. EuroSys 2017*, pp. 286–297, 2017.
[15] Philip Carinhas, **Linux Fundamentals - A Training Manual**, *Fortuitous Technologies Inc*, 2001.
[16] Intel Corporation, **Intel® 64 and IA-32 Architectures Software Developer Manuals**, vol 3C, 2016.
[17] Y. Ishida, **Self-Repair Networks - A Mechanism Design**, *Springer (Switzerland)*, volume 101, 2015.
[18] M. Cohen, **Scanning Memory with Yara**, *Digital Investigation*, volume 20, 2017.
[19] S. Proskurin, T. Lengyel, M. Momeu, C. Eckert, and A. Zarras, **Hiding in the shadows: Empowering arm for stealthy virtual machine introspection**, *Proceeding of ACM International Conference*, pp. 407-417, 2018.

[20] N. Smyth, **Xen Virtualization Essentials**, *Payload Media*, Ed. 1, pp. 124-125, 2009.
[21] I. Winarno, M. Sani, **Automatic Backup System for Virtualization Environment**, *EMITTER International Journal of Engineering Technology*, vol. 2, pp. 91-101, 2014.