ICIC Express Letters
Volume **17**, Number **11**, November **2023**
ICIC International ©2023 ISSN 1881-803X
pp. 1195–1207

ENABLING A CENTRALIZED LOGGING SYSTEM IN AN INFRASTRUCTURE AS A SERVICE CLOUD

Wichep Jaiboon[2], Pakorn Chan-in[3], Winai Wongthai[1,2,*] and Thanathorn Phoka[1,2]

[1]Research Center for Academic Excellence in Nonlinear Analysis and Optimization
[2]Department of Computer Science and Information Technology
Faculty of Science
Naresuan University
99 Moo 9, Phitsanulok-Nakhonsawan Road, Tambon Tapho, Muang, Phitsanulok 65000, Thailand
{ wichepj59; thanathornp }@nu.ac.th; *Corresponding author: winaiw@nu.ac.th

[3]Department of Computer Science
Faculty of Science and Agricultural Technology
Rajamangala University of Technology Lanna Nan
59 Moo 13, Tambon Fai Kaeo, Amphoe Phu Phiang, Nan 55000, Thailand
pakornc@rmutl.ac.th

Received December 2022; accepted March 2023

Abstract. *This paper proposes a new architecture to enable a centralized logging system in an Infrastructure as a Service cloud. The main contribution of this work can improve the implementation and management of the current decentralized logging systems. This is where only a socket programming method is required as a primary method of the improvement. Achieving the new architecture is by redesigning the architecture of our own existing logging system, which was already in an OpenStack environment. Based on the patterns and behaviours of this environment, the redesigning process mainly includes relocating the main components of the logging system and generating a new architecture to enable a centralized logging system in this cloud type. To examine the successes of our new architecture, the OpenStack environment is also the main technology to be used to implement a logging system based on the new architecture and for the experiments. Then, this logging system is used to test the main logging tasks, including detecting malicious processes inside a customer cloud. The results of the tests show that we can successfully detect malicious processes inside the customer cloud using the logging system of the new-centralized architecture. The logging systems with the previous-decentralized architecture can also detect this kind of process, but with more than one manager, compared to only one manager of the new architecture. Thus, with only one manager, the centralized architecture could be easier to be implemented and be managed than the decentralized one. As a central contribution to the cloud security field, this centralized architecture can help in mitigating the cloud security issues, such as malicious process activities inside the cloud. The issues are the critical one that prevents the adoption of the cloud.*
Keywords: Cloud security, Infrastructure as a Service, Centralized logging system, OpenStack, Malicious process

1. **Introduction.** Cloud computing or the cloud offers visualized computing, storage, and networking resources over the Internet to organizations or customers. The cloud service rental model is flexible. The model allows easily reducing, adding, or discontinuing the services. This makes the cloud to be interesting in organizations bringing it to their IT department. This paper focuses on the Infrastructure as a Service or IaaS cloud type. IaaS mainly offers rentable Virtual Machines (VMs) to consumers. However, cloud security is a critical issue that prevents the adoption of the cloud. The issue has been considered

DOI: 10.24507/icicel.17.11.1195

by many researchers. This is especially about 14 years-consideration in about 23 reports with five languages of the Cloud Security Alliance (CSA). An example is the provision of the 'Top Threats to Cloud Computing, version 1.0' report [1] in 2010 to 'Top Threats to Cloud Computing Pandemic Eleven' in 2022 [2]. Researchers have also been investigating how to prevent and mitigate the risks associated with the issue. The results from the investigation can enhance the confidence of consumers or organizations that want to adopt the cloud. In an IaaS cloud, one of the mitigating solutions is a logging system. This system can help in mitigating risks associated with the issue, as discussed in [3, 4, 5, 6]. This system facilitates investigating suspicious events, for example, who is accessing a cloud customer's sensitive file in an IaaS environment. Thus, the system can detect and report what unauthorized users do with the file.

Motivation: Thus, this file has its own history produced by this logging system. This system can be categorized as a file-centric logging system rather than a system-centric one that only logs a machine's health data, such as the total percentage of CPU usage, not a sensitive file's history. A file-centric log can be an event of, for example, which file is accessed by which process. Oppositely, this can be seen as a process-centric log perspective as to which process is accessing which file. Both file-centric and process-centric logs can be produced by a Virtual Machine Introspection (VMI). It can introspect into a main memory of a VM to capture a process's information and behaviours. In recent years, VMI has been useful for Intrusion Detection Systems (IDS) in cloud computing as agreed and implemented by [7, 8]. However, the systems from both references are decentralized. [9] states that it is tough to manage the decentralized log-files of a logs management in cloud computing. We also believe that the logger components of these decentralized systems are difficult to be managed. Thus, in an IaaS cloud, our motivation is to enable logger components of a logging system to be centralized. This system can be considered a new logging system architecture in IaaS. Our paper aims to expand the capabilities of existing logging systems. To have efficient recording capabilities of the events in the IaaS cloud, the study of the functions of the logging system can be vital.

However, the previous works [4, 5, 6] provide the logging systems in a laboratory IaaS environment rather than in a production IaaS cloud environment. Thus, this paper focused on enabling the existing logging system in [4, 5, 6] to be able to work in the production IaaS cloud environment. We simulated this environment with a cloud Operating System (OS) called OpenStack [10]. This cloud OS has increasingly been applied in a cloud production business [11, 12]. Then, we used the socket programming method [13] to enable the existing logging system to work in this simulated environment. The socket programming method is a technique for exchanging messages between processes on the same computer or across a network. This method can reside on the same system or different systems on a different network. Finally, in our previous work [3], we managed to enable the existing logging system to work in the simulated production IaaS cloud environment. This enabled logging system can help in mitigating risks associated with the cloud issue in this environment. In [3], one of the main components of our architecture is a compute node. It is a physical computer/machine with hypervisor software. The software can create more than one Virtual Machine (VM) in this physical machine. A logging system is a kind of monitoring system that can be distributed into many compute nodes. This distribution can cause difficulty in the management of the nodes.

Research gaps and objectives: There are three research gaps and three objectives. Firstly, in the OpenStack architecture, there can be more than one compute node in the architecture, as briefly discussed above and thoroughly discussed in [3]. Thus, there can be more than one logging system in these compute nodes of this architecture. This is a distributed/decentralized architecture (such as the ones of [3, 7]) for the logging system. This decentralized architecture is difficult to be managed based on our primitive experiments of enabling logging systems into the OpenStack architecture. Thus, the first objective is that we aim to enable the existing logging system from [3] to work in an OpenStack architecture by using the socket programming method. This enables a centralized architecture for the logging system, which could be easier to be managed. From the first research gap, the second gap is that there are no design and implementation of the logging system using the socket programming method in the OpenStack architecture in the literature. Thus, the second objective is that we will provide both the design and implementation. The third gap is that there are no results and discussions to illustrate how the results from the design and implementation work in the OpenStack architecture. Then, the final objective is to provide these results and discussions.

Summary of contributions: This paper has the following four contributions. The first is that we discuss an IaaS cloud and how the existing logging system in the OpenStack architecture works. Then, we illustrate that the existing logging system from [3] is a decentralized architecture and challenging to manage in the OpenStack architecture. The second contribution is that, based on the first contribution, we redesigned the existing logging system to support centralized architecture via the socket programming method. This includes designing and implementing the logging system in the OpenStack architecture of an IaaS cloud. This design and implementation ensure the technical details of redesigning the existing logging system to apply in the OpenStack architecture. The third one is that we illustrate the results from the design and implementation of the second contribution. The results confirm that the redesign of the existing logging system can still detect malicious processes like our previous logging system but with easier management compared to the previous system. The last contribution is the discussions of the design and implementation and the results. The examples of the discussions are why the redesign of the existing logging system can still work in the real-world IaaS cloud, which is built from OpenStack or an IaaS-OpenStack cloud, and this logging system in the IaaS-OpenStack cloud can be easier to manage than the previous existing logging system.

The organization of this paper is the following. The background is described in Section 2, including IaaS architecture, the existing logging system architecture in the laboratory IaaS cloud environment, an OpenStack architecture, and the architecture of the logging system in the OpenStack for IaaS. The system design and implementation of the proposed architecture are discussed in Section 3, including the hardware and software of the experimental environment, the aim of the new logging system architecture based on the socket programming method for IaaS, and the proposed new logging system architecture in IaaS-OpenStack cloud. The results and discussions of the proposed architecture are in Sections 4 and 5, respectively. Section 6 includes the conclusion and future work of this paper.

2. **Background.**

2.1. **Infrastructure as a Service architecture.**

Figure 1 shows an IaaS cloud and logging system architectures. In this paper, both architectures are mainly adapted from our previous work [4]. All the white boxes, an ellipse, and text file shape in Figure 1 are the components of the IaaS cloud architecture. This section will describe these components. All the shaded boxes in Figure 1 are the logging system's main components. Section 2.2 will describe these main components. This section here will shortly explain the IaaS cloud architecture as the following. The white box components include hypervisor, dom0, hw0, disk0, domU, hwU, diskU, and memU. A component name ending with '0' implies that this component is physically owned and managed by an IaaS cloud provider. However, ending with 'U' implies that the component is virtually owned and managed by a cloud customer. See the box with number 2 in Figure 1, and it is a hypervisor or software that grants a physical computer to host more than one VM.

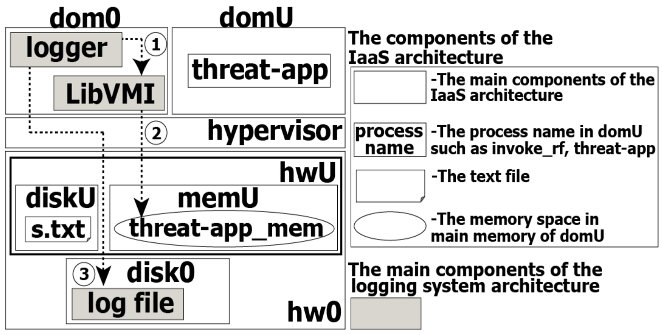

dom0
domU
logger
1
LibVMI
threat-app
2
hypervisor
hwU
diskU
s.txt
memU
threat-app_mem
3
disk0
log file
hw0
The components of the IaaS architecture
-The main components of the IaaS architecture
process name
-The process name in domU such as invoke_rf, threat-app
-The text file
-The memory space in main memory of domU
The main components of the logging system architecture

FIGURE 1. The IaaS cloud architecture and logging system architecture, adapted from [4]

The box inside the domU, the ellipse in memU, and the text file shape in Figure 1 are the components for the simulation of some incidents in our experimental purposes in Section 3. A dom0 or domain 0 is a manager of the entire VMs that were created by the customers; see the topmost left white box in Figure 1. This domain is also a VM and is launched by the hypervisor at the system booting time. It completely accesses and manipulates hw0 and all the created VMs or domUs. Hw0 is the physical hardware of and managed by a dom0; see the bottom box in Figure 1. A domU or user domain is a user VM that is created by dom0; see the topmost right white box in Figure 1. It runs on top of the hypervisor and is an IaaS cloud product that the provider can rent to the customers. HwU is physically located in hw0 and is the virtual hardware of a domU. It is physically owned and managed by a dom0 or by the provider. However, it is virtually owned and managed by a domU owner or a customer. A disk0 is a physical disk of a dom0, and a diskU is a virtual disk of a domU. Finally, memU is domU's virtual main memory.

2.2. **The existing logging system architecture in laboratory IaaS cloud environment.** The IaaS cloud architecture and logging system architecture in Figure 1 are in a non-real-world production environment. This environment was built on one computer machine to simulate an IaaS cloud in [4, 5, 6, 14]. The existing logging system architecture in Figure 1 can log incidents that happened in a customer domU or VM, such as who has access to or what happens with a customer file in a disk of the VM [14, 15]. A logging system can compose of a logging process and a log file [14]. This paper will call the logging process a logger. The system architecture of the logging system is from our previous work [14] and is also illustrated in Figure 1. In this paper here, the architecture will be applied to the proposed experiment. The box inside the domU in Figure 1 is the threat-app process. For the purposes of the experiment, we assume that, somehow, this process can be controlled by an attacker. As a result, he/she can maliciously read a sensitive file or s.txt of an IaaS customer in diskU, and see the text file shape inside the diskU. A threat-app_mem in memU represents a reserve memory space for the threat-app process provided by the Operating System (OS) that hosts this process. The right bottom shaded box in the dom0 is LibVMI, the new name of XenAccess [16]. It is a C library that is installed in the dom0. Then, it can access the threat-app_mem in memU to detect the malicious activities of the threat-app process that is reading s.txt. We did not focus on this reading s.txt action in this paper.

However, the action is also referred to in proposing logging solutions in the cloud. In [15, 17] consider this action when proposing their logging solutions in the cloud. For example, [15] assumes that when a user 'Alice' creates a sensitive file (such as s.txt) and modifies this file. Then, somehow, a user, 'Bob', can maliciously read the file without Alice's permission. From Figure 1, the three working steps of the logger are represented by the circles with the numbers 1 to 3. Step 1, the logger in the dom0 calls LibVMI to access memU to obtain the logging data from the threat-app_mem (Step 2). This data includes i) a file name of s.txt or the string ‘s.txt’, and ii) the process ID of the threat-app process. Then, LibVMI accesses memU to obtain this data in the threat-app_mem. Then, it returns the obtained data to the logger. Finally, the logger manages the data and then writes (Step 3) the data into the log file.

2.3. **An OpenStack architecture.** The IaaS cloud architecture and logging system architecture in Figure 1 are in a non-real-world production environment. This environment was built on one computer machine to simulate an IaaS cloud in [4, 14]. We will use OpenStack to simulate an IaaS public cloud to be the same as a real-world IaaS production environment. OpenStack is a cloud operating system that controls larger pools of storage, networking, and compute resources in all parts of a data center. The resources are managed via a dashboard that is a web interface [18]. OpenStack is also open-source software that can be used to build an IaaS public cloud, and it can be used in the real-world cloud production environment. It has increasingly been used in large business organizations [19]. Thus, we use OpenStack to simulate an IaaS public cloud to be the same as a real-world IaaS production environment. See the five boxes in Figure 2. There are five services of the architecture of an IaaS-OpenStack cloud. These services are the compute service, image service, object storage service, dashboard service, and identity service. All of these services are connected through the communication network; see the solid lines in Figure 2. The details of each service are mainly from the OpenStack official documentation website [20]. This paper relates to only three services. The first service is the compute service; see the box on the leftmost of Figure 2. The compute service can enable a provider to control an IaaS cloud. This service allows the provider to control instances (VMs/domUs) and networks and to handle access to the cloud by users. This service also defines drivers that interact with a hypervisor that is run on the host OS or dom0 of the provider cloud infrastructure. This service also publishes its functionality for other services via web-based OpenStack Application Program Interfaces (APIs). The second is the object storage service, the box on the rightmost of Figure 2. It is a long-term storage system for many static data, which can be updated and retrieved. The last service is the box on the top of Figure 2. It is the dashboard service, which is a web-based interface allowing the provider to manage OpenStack resources and services. It also interacts with the compute service via the APIs.

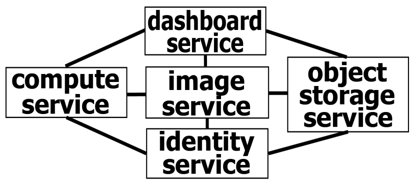


Figure 2. An OpenStack architecture from [3]

2.4. **The existing architecture of the logging system in the OpenStack for IaaS.** The architecture of the logging system in the OpenStack for IaaS was from our previous work [3] to prove that the logging system (Figure 1) can work in the OpenStack environment (Figure 2). In Figure 3, we illustrated the perspective of ‘the architecture of the logging system in the OpenStack for IaaS’. We will call this architecture the ‘logging system in IaaS-OpenStack cloud’. However, this figure without the logging components (logger, LibVMI, and log file) is called ‘an IaaS-OpenStack cloud’. We run Ubuntu Server 16.04 LTS OS on a computer or hw0 as the compute node. Then we installed the OpenStack Ocata version in this OS. We mapped each service, except the object storage service, of the OpenStack architecture in Figure 2 onto two Physical Machines (PMs).

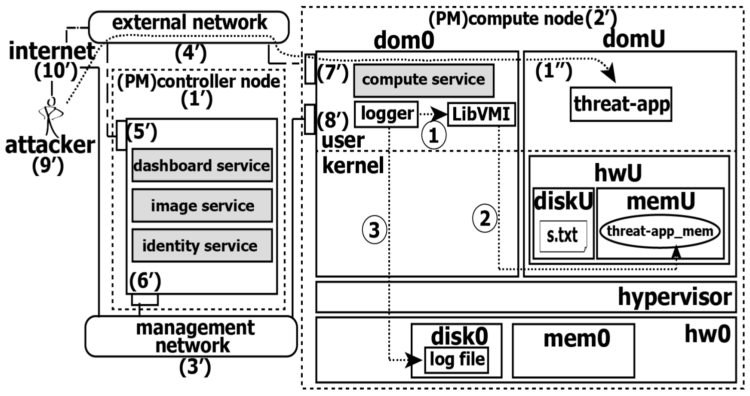


FIGURE 3. The architecture of the logging system in the OpenStack for IaaS, adapted from [3]

According to the functions in the installation manual of the OpenStack [21], one of the two physical computers is the compute node, and another one is the controller node. The dotted box with the number $1'$ on the left side of Figure 3 is the controller node for the execution of the three services of Figure 2. The services are the identity service, image service, and dashboard service; see the shaded rectangles in the dotted box. These services were described in Section 2.3. Then, the dotted box with the number $2'$ on the right side of Figure 3 is the compute node for the execution of the compute service from Figure 2. This service is used to control the hypervisor to manage dom0 and domUs. Both of the nodes use two network adapters ($5'$ to $8'$) to connect to two networks. They are the management network ($3'$) and the external network ($4'$). The management network is used for installing applications and for updating the patch of the security of the OSes of the controller node and the compute node. In this IaaS-OpenStack environment, the external network is used for the customers to connect to domUs via the Internet or $10'$. In the architecture, this external network may also be somehow a channel that the attacker or $9'$ can use this channel to connect to domUs with the credentials and passwords that the attacker may steal through phishing and fraud as argued in [14], see the dotted line with number $1''$. Now, we got our IaaS-OpenStack cloud.

Then, we simulated incidents that were assumed to be taken place by the attacker. Thus, the attacker can connect to a domU in this built IaaS-OpenStack cloud over the Internet using the credentials and password mentioned above. See the box in the user level of domU in Figure 3. Then, the attacker may use a 'threat-app' process to access the sensitive file in diskU. We aim to locate the existing logging system or the logger (see the logger box in the user level of dom0) and the log file in Figure 1 into this built IaaS-OpenStack cloud. So, this logger has been designed to detect a threat-app application (threat-app process). We consider the threat-app process as a target process that the logger wants to capture the appearance of this process for the detection of such incidents. Now, we got the architecture of the logging system in the OpenStack for IaaS, adapted from [3], see Figure 3. Then, this architecture can detect the threat-app process, as discussed in [3].

The IaaS-OpenStack cloud architecture can expand many compute nodes in the real-world IaaS production environment, as discussed in [3, 22] that have illustrated the expanding of the compute nodes as 1,000 nodes in an OpenStack architecture. The expansion can be done by adding other compute nodes into the architecture of the IaaS-OpenStack cloud in Figure 3. All new nodes' details are the same as a compute node in the architecture of the IaaS-OpenStack cloud. The new nodes can be set up based on OpenStack Installation Guide 15.0 in [21]. Thus, there can be more than one logger in these compute nodes, and executing the logger in each compute node is difficult to manage. Therefore, we will call this logging system in Figure 3 a decentralized logging system. Thus, we will enable a centralized architecture for a logging system in the IaaS-OpenStack cloud architecture or a centralized logging system. It is our primary objective.

3. Design and Implementation.

3.1. **The hardware and software of the experimental environment.** This section will explain the experimental environment by expanding the information of Figure 3, just discussed in Section 2.4. This can form the experimental IaaS-OpenStack cloud. We then set up this cloud using two PMs. The first PM composes of Intel Core-i7 3.2 GHz 64-bit four cores, DDR3-SDRAM 8 GB of main memory, and 500 GB of secondary storage. This PM is the controller node; see number $1'$ in Figure 3. The controller node is configured with the default settings based on the OpenStack Installation Guide 15.0 [21]. The compute node is based on another PM of Intel Xeon 3.4 GHz with CPU 64-bit eight cores. This PM has DDR3-SDRAM 8 GB of main memory and 500 GB of secondary storage; see number $2'$ in Figure 3. A Ubuntu Server 16.04 LTS is the OS of both PMs. Then, we installed Xen 4.6.5 as the hypervisor in this compute node. This creates the dom0 in the figure. We also installed the LibVMI library (version 0.12-rc3), the shaded box inside the dom0 in Figure 5, in the compute node. Finally, in the compute node, we then created the domU in Figure 3, with 1 CPU core, 1 GB of memU, and 20 GB of diskU.

3.2. **The aim of the new logging system architecture based on the socket programming method for IaaS.** The new logging system architecture can do the centralized architecture for a logging system in the IaaS-OpenStack cloud based on the socket programming method for IaaS. This centralized architecture needs to modify the existing logging system in Figure 1. From the first research gap in Section 1 that the existing logging system in Figure 3 is a decentralized architecture. This architecture is difficult to manage, and this gap. It is because the IaaS-OpenStack cloud architecture in Figure 3 can be more than one compute node in the real-world IaaS production, as discussion in [3]. Thus, there can also be more than one logging system in these compute nodes.

3.2.1. *The components and parameters.* We will redesign the existing logging system in Figure 1 in Section 2.2 to support centralized architecture via a socket programming method and based on Figure 2 and Figure 3. The socket programming method or socket method is one of the best methods in distributed computing, as argued in [23]. Figure 4 shows the new logging system architecture based on the socket programming method for IaaS. This figure is proof that all the existing logging components (LibVMI and log file) and the new-modified components (logger_c and logger_s) can be fitted in the IaaS architecture (or Figure 1). The architecture in Figure 4 is split into two physical machines. We call the first one a 'server-side', and the second one a 'client-side'. All the components

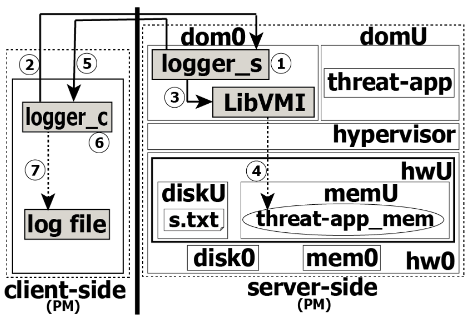


FIGURE 4. The new logging system architecture based on the socket programming method for IaaS, not in IaaS-OpenStack cloud

and their functionalities on the server-side are the same as the ones in Figure 1, except the logger_s. The logger_s in Figure 4 is different from the logger in Figure 1. It is because we added the socket method into the logger_s. This enables logger_s to exchange messages between itself (in a server-side physical machine) and logger_c (in a client-side physical machine) across a network. A client-side is a physical machine with the components of logger_c and log file. The details of both are below. The client-side machine and the server-side machines can communicate via a Local Area Network (LAN). The main logging components of the new logging system based on the socket method for IaaS are all gray boxes in Figure 4. They are logger_s, log file, logger_c, and LibVMI. A logger_s has been designed to detect threat-app processes in domU. It works the same as the logger in Figure 1. However, the logger_s is designed to receive a connection from logger_c from the client-side to exchange the data, such as domU's name and process name of the target process, such as threat-app. See the gray box on the top side of the client-side in Figure 4; it is logger_c that is designed to request and then connect to logger_s in the server-side. See the gray boxes that are labeled with "LibVMI" and "log file". Both components were functioning as Figure 1 and were described in Section 2.2. As mentioned above, Figure 4 shows a new logging system architecture based on the socket programming method for IaaS or centralized logging system architecture.

3.2.2. *The working steps of the new logging system architecture.* There are seven working steps of the new logging system architecture based on the socket method for IaaS or centralized logging system architecture. Step 1 is when the logger_s was started and is waiting for a connection request from the logger_c. Step 2 is when the logger_c is successfully connected to the logger_s; then, the logger_c sends a message to the logger_s. The message includes i) a target domU's name in the server-side machine (such as 'du1') and ii) the target process's name in the domU (such as 'threat-app'). Step 3 is when the logger_s gets the message from the logger_c, and logger_s uses the message to set up the initial parameters for calling LibVMI. Then, Step 4 is when LibVMI retrieves appropriate logging data from the memU of the target domU based on the parameters. Step 5 is after the logger_s gets the appropriate logging data (the process' id and process' name of threat-app), the logger_s then sends this data to the logger_c. Step 6 is when the logger_c gets the data (message) from Step 5. The logger_c then pre-processes the data before writing the processed data to the 'log file' in Step 7.

3.3. **The proposed new logging system architecture in IaaS-OpenStack cloud with expanding the compute nodes.** Figure 5 without all the shaded boxes shows the IaaS-OpenStack cloud with expanding the compute nodes. The IaaS-OpenStack cloud

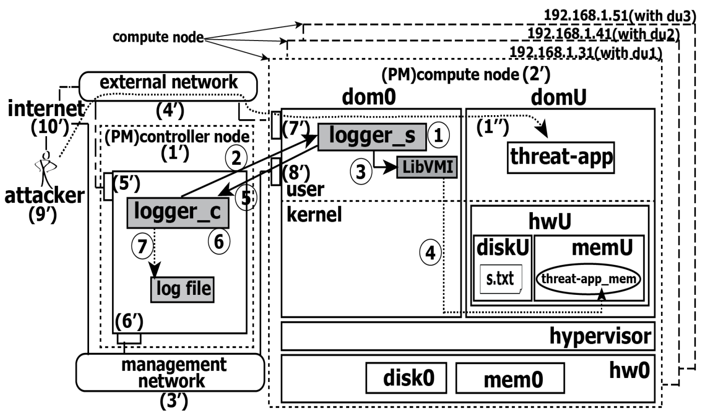
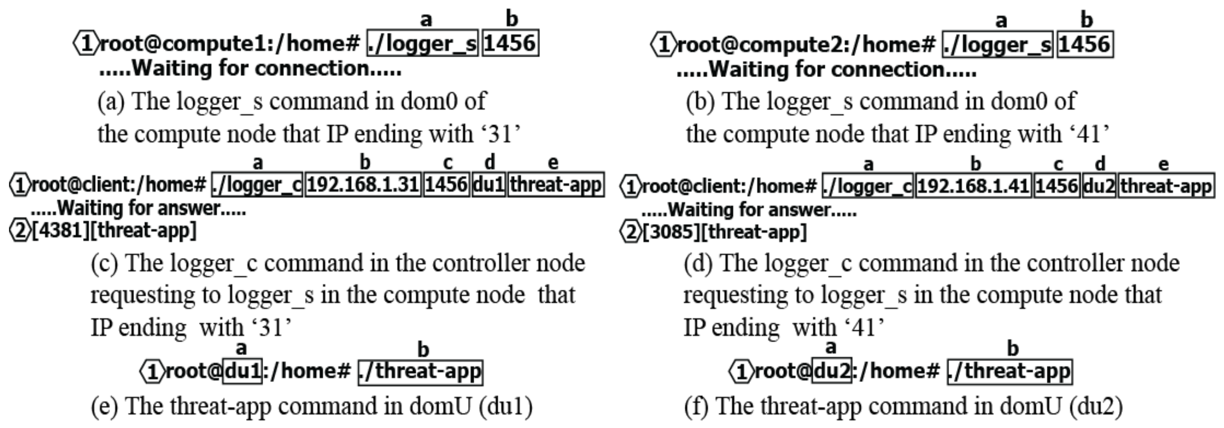


FIGURE 5. The proposed centralized architecture for a logging system in IaaS-OpenStack cloud

architecture and the expanding compute nodes were described in Section 2.4. In this experiment, we add the two additional compute nodes into the IaaS-OpenStack cloud in Figure 3. Thus, on the right side of Figure 5, expanding compute nodes are the dashed boxes with IP addresses ending with 41 and 51. The proposed new logging system architecture in the IaaS-OpenStack cloud with expanding the compute nodes is all the shaded boxes in Figure 5. This logging system has the main components and the location of these components as Figure 4. The dotted box with the number $1'$ on the left side of Figure 5 is the controller node as a physical computer in the IaaS-OpenStack cloud. First, we map the two components and location of the new logging system architecture in Figure 4, including logger_c and log file into the controller node in Figure 5. Then we map the component and location of logger_s and LibVMI in Figure 4 into the compute node; see the dotted box with the number $2'$ on the right side of Figure 5. Furthermore, each expanding compute node in Figure 5 has a logger_s, a LibVMI component, and the location of these components in side itself as the logger_s and LibVMI in the dotted box with the number $2'$ on the right side of Figure 5. The detail of the main components of the proposed new logging system architecture in the IaaS-OpenStack cloud with expanding the compute nodes is described in Section 3.2.1. Now, we got the proposed new logging system architecture in the IaaS-OpenStack cloud with expanding compute nodes.

4. **The Results.** The operational results of the proposed new logging system architecture of Figure 5 are shown in Figure 6 (Figures 6(a) to 6(f)). The figure shows the command line patterns of logger_s, logger_c, and threat-app, and the commands' results. The first part of the results are Figures 6(a) to 6(d), which were mainly generated from the six working steps of the new logging system architecture in Section 3.2.2. The second part of the results is Figures 6(e) to 6(f). They were mainly generated from the working steps of the threat-app, discussed in Section 2.2. In Figure 6(a), the line with number 1 is a command line pattern of the logger_s when it wants to capture the name and Id of the target monitored process or thread-app on the IaaS customer's domU. The logger_s in the box with 'a' has one parameter as the port number in the box with 'b' or 1456. This command is executed in dom0 by an authorized user. After the logger_s is executed, it will be waiting for a connection request from logger_c. Logger_c needs to be placed on a PM that connects with the same management network (see $3'$ in Figure 5) of the IaaS-OpenStack cloud. In Figure 6(c), the line with the number 1 is a command line pattern of the logger_c in the box with 'a'. This logger_c works with the logger_s together when it wants to monitor the target process or threat-app on the IaaS customer's domU. The logger_c has four parameters: Internet Protocol address (IP address), port number, domU's name, and target process's name. The IP address in the box with 'b' or '192.168.1.31' is a unique address that identifies a device on a local network of a compute node in the IaaS-OpenStack cloud. The port number in the box with 'c' or '1456' identifies a port of logger_s service on the compute node in the IaaS-OpenStack cloud. This port number needs to be the same number in the box with 'b' of Figure 6(a).

The domU's name in the box with 'd' or 'du1' is a target domU's name in the compute nodes in the IaaS-OpenStack cloud. Finally, the target process's name in the box with 'e' or 'threat-app' is a target process's name in the domU. The logger_s wants to monitor this name. The cooperation working steps between the logger_s and the logger_c were described in Section 3.2.2. Then, Figure 6(d) has the same command line patterns as Figure 6(c). The differences can be the parameters of boxes 'b', 'c', 'd', and 'e'. For example, Figure 6(d) has two different parameters from Figure 6(c). See boxes 'b' and 'd' of Figure 6(d); they are the IP ending with 41 (instead of 31) and du2 (instead of du1). Figure 6(e) is when the threat-app command is executed in domU or, see the line with number 1 in the figure. Then, the name and the Id of the threat-app appeared in the reserve memory space inside du1's memU. This space is called threat-app_mem; see the oval shape in



```
①root@compute1:/home# ./logger_s 1456
.....Waiting for connection.....
```

(a) The logger_s command in dom0 of the compute node that IP ending with '31'

```
①root@compute2:/home# ./logger_s 1456
.....Waiting for connection.....
```

(b) The logger_s command in dom0 of the compute node that IP ending with '41'

```
①root@client:/home# ./logger_c 192.168.1.31 1456 du1 threat-app
.....Waiting for answer.....
②[4381][threat-app]
```

(c) The logger_c command in the controller node requesting to logger_s in the compute node that IP ending with '31'

```
①root@client:/home# ./logger_c 192.168.1.41 1456 du2 threat-app
.....Waiting for answer.....
②[3085][threat-app]
```

(d) The logger_c command in the controller node requesting to logger_s in the compute node that IP ending with '41'

```
①root@du1:/home# ./threat-app
```

(e) The threat-app command in domU (du1)

```
①root@du2:/home# ./threat-app
```

(f) The threat-app command in domU (du2)

FIGURE 6. The command line patterns of logger_s, logger_c, and threat-app, and the commands' results

Figure 1, already discussed in Section 2.2. Then, the logger_s (in Figure 6(a)) can capture the Id of threat-app as '4381' and the name of threat-app as 'threat-app'; see the line with number 2 of Figure 6(c). This is the same mechanisms with Figures 6(f), 6(b), and 6(d). Note that, in our experiment environment, the 'threat-app' in Figure 6(e) and the one in Figure 6(f) have the same name. However, they are different malicious applications with different process Ids in different domUs (du1 and du2). Figure 6(d) illustrates that the logger_c is in the same physical computer logger_c in Figure 6(c). Logger_c in Figure 6(c) and Figure 6(d) is the same one and is placed in the same physical computer. This computer is called the controller node; see the dotted box with the number $1'$ on the left side of Figure 3. Thus, the proposed new logging architecture in the IaaS-OpenStack cloud with expanding the compute nodes in Figure 5 is a centralized logging system. This is because there is only one logger_c in the architecture. Then, this logger_c can make requests to, for example, two logger_s(s). One logger_s is in box 'a' of Figure 6(a), and another one is also in box 'a' of Figure 6(b), just discussed above.

5. Discussions.

5.1. **Easier management of the centralized compared to decentralized architectures.** This discussion is against the three research gaps discussed in Section 1. To be clearer for the discussions, we modified Figure 3 to become Figure 7. From Figure 7, we can see that there are three PMs with three IPs ending with 31, 41, and 51. A logger (see the shaded box in the figure) is actually placed in each of the PMs. Thus, there will be three loggers for all the PMs. Consequently, these loggers are distributed (we called 'decentralized') into the whole architecture of Figure 7. The authorized user needs to configure and manage each logger one by one in this decentralized architecture. This should be difficult management and discussed in the introduction section as the main researcher gap. In Figure 5, there are actually also three loggers (we called each one 'logger_s') for all PMs, with three IPs ending with 31, 41, and 51. However, all these three logger_s(s) are managed by only one manager (called logger_c). Thus, this manager is not distributed (we called it 'centralized') into the whole architecture of Figure 5. The authorized user can only configure and manage all the logger_s with one point at the logger_c in this centralized architecture. This should be easier management, compared to Figure 7.

Moreover, based on the results in Section 4, Figures 6(a) to 6(f) illustrated the results of the implementation of the design of this centralized architecture in Figure 5. Especially, Figures 6(c) and 6(d) show that only one manager, which is logger_c, in boxes with 'a' from both figures. From the results in Section 4, this manager can manage two logger_s(s) to record/log the existence of the malicious processes in the two PMs, with two IPs ending with 31 and 41. An example of these processes is the 'threat-app' (see box 'b' in

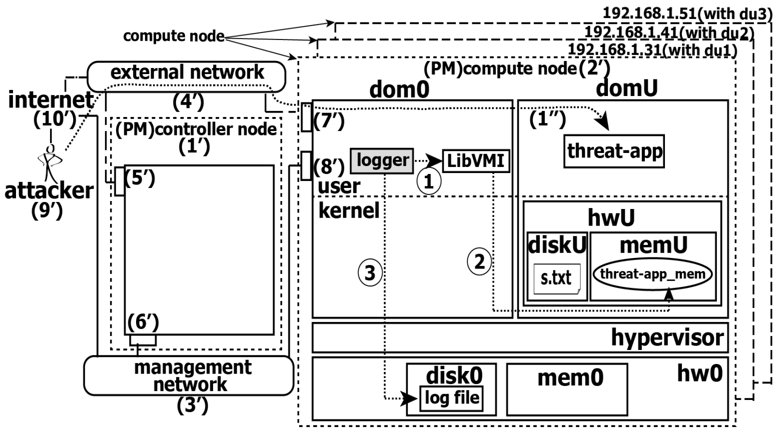


FIGURE 7. The existing decentralized architecture for a logging system in IaaS-OpenStack cloud

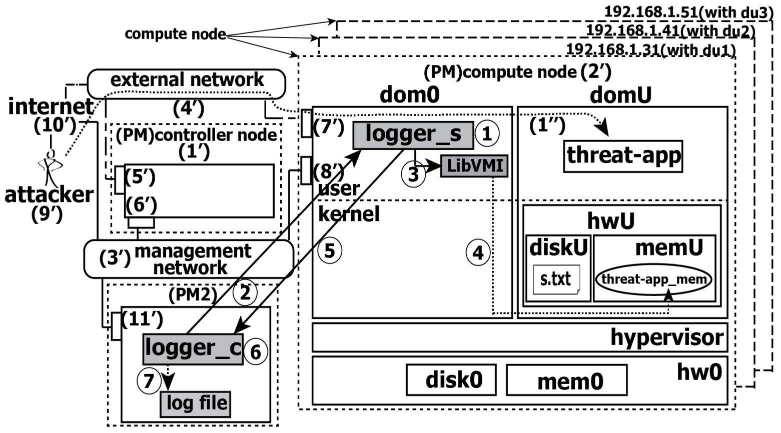


FIGURE 8. Separation of the workloads of the controller node

Figure 6(e)). This application is in domU called du1; see du1 in the box 'a' in Figure 6(e). Another example of these malicious processes is also the 'threat-app' (see box 'b' in Figure 6(f)) in du2 (see box 'a' in Figure 6(f)). To sum up, our proposed centralized architecture for a logging system in the IaaS-OpenStack cloud could be mitigated the main research gap, which is the difficulty of the management of the decentralized loggers in this cloud. We believe that it could be easier to manage centralized compared to decentralized architectures.

5.2. **Separation of the workloads of the controller node.** From Figure 5, we can see that the controller node is one PM (we will call PM1; see number $1'$ in Figure 5). This PM1 has the logger_c and log file (the shaded boxes in the controller node in Figure 5). We can actually relocate the logger_c and log file to place into the new available PM2; see the left-lower dotted box in Figure 8. This relocation can separate the workloads of the controller node, or PM1, from the workloads of the logging jobs. Thus, PM1 can perform only its own busy necessary main jobs (such as dashboard, image, and identity service). Now, this PM1 has no management of the logging jobs with the logger_c and log file anymore. Briefly, this separation can be done by the two main steps. The first one is to move the logger_c and log file into PM2. The second step is to connect the PM2 to PM1's management network or to number $3'$ in Figure 5. The results of this connection are the management network or number $3'$ in Figure 8. The network allows PM2 to be connected to or worked with the other components of our proposed centralized architecture, shown by Figure 8. This enables PM1 and PM2 to separately perform their own native necessary tasks. To sum up, the final results of both steps are in Figure 8, which can be the separation of the workloads of the controller node. This could enhance the overall system management and performance of this architecture.

6. **Conclusions.** This paper aims to enable a centralized logging system in an Infrastructure as a Service (IaaS) cloud on a cloud Operating System (OS) or environment. This paper focused on the OpenStack environment. The experiments confirmed that we successfully enabled the centralized logging system in this IaaS cloud. The results from the experiments indicate that we can use the socket programming method to allow us to redesign the architecture of and relocate the main components of the existing logging systems. This enabled the decentralized and difficult-management logging system to be the centralized and easier-management logging system or 'a centralized logging system'. This paper makes the main contribution to the field of cloud security, which is a critical issue that prevents the adoption of the cloud. As discussed by and shown in Figure 8, the future work is to implement the separation of the workloads of the main physical machine (the controller node) in our centralized logging system.

REFERENCES

[1] CSA, *Top Threats to Cloud Computing, Version 1.0*, The Cloud Security Alliance (CSA), Tech. Rep., 2010.

[2] CSA, *Top Threats to Cloud Computing Pandemic Eleven*, The Cloud Security Alliance (CSA), Tech. Rep., 2022.

[3] W. Jaiboon, W. Wongthai, T. Phoka and T. Auxsorn, A logging system in OpenStack environment to mitigate risks associated with threats in Infrastructure as a Service cloud, *ICIC Express Letters*, vol.14, no.4, pp.387-397, 2020.

[4] S. Wiriya, W. Wongthai and T. Phoka, The enhancement of logging system accuracy for Infrastructure as a Service cloud, *Bulletin of Electrical Engineering and Informatics*, vol.9, no.4, pp.1558-1568, 2020.

[5] T. Auxsorn, W. Wongthai, T. Porka and W. Jaiboon, The accuracy measurement of logging systems on different hardware environments in Infrastructure as a Service cloud, *ICIC Express Letters, Part B: Applications*, vol.11, no.5, pp.427-437, 2020.

[6] W. Wongthai, F. L. Rocha and A. van Moorsel, A generic logging template for infrastructure as a service cloud, *2013 27th International Conference on Advanced Information Networking and Applications Workshops*, pp.1153-1160, 2013.

[7] P. Mishra, I. Verma and S. Gupta, KVMinspector: KVM based introspection approach to detect malware in cloud environment, *Journal of Information Security and Applications*, vol.51, 102460, 2020.

[8] A. A. R. Melvin, G. J. W. Kathrine, S. S. Ilango, S. Vimal, S. Rho, N. N. Xiong and Y. Nam, Dynamic malware attack dataset leveraging virtual machine monitor audit data for the detection of intrusions in cloud, *Transactions on Emerging Telecommunications Technologies*, vol.33, no.4, e4287, 2022.

[9] V. Agrawal, D. Kotia, K. Moshirian and M. Kim, Log-based cloud monitoring system for OpenStack, *2018 IEEE 4th International Conference on Big Data Computing Service and Applications (BigDataService)*, pp.276-281, 2018.

[10] O. Sefraoui, M. Aissaoui, M. Eleuldj et al., OpenStack: Toward an open-source solution for cloud computing, *International Journal of Computer Applications*, vol.55, no.3, pp.38-42, 2012.

[11] ubuntu.com, *OpenStack Is Dead? The Numbers Speak for Themselves*, 2022.

[12] OpenStack.org, *OpenStack Survey Report*, 2022.

[13] ibm.com, *IBM i Version 7.2 Programming Socket Programming*, 2013.

[14] W. Wongthai, *Systematic Support for Accountability in the Cloud*, Ph.D. Thesis, Newcastle University, 2014.

[15] R. K. L. Ko, P. Jagadpramana and B. S. Lee, Flogger: A file-centric logger for monitoring file access and transfers within cloud computing environments, *IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, 2011.

[16] B. D. Payne, M. D. P. D. A. Carbone and W. Lee, Secure and flexible monitoring of virtual machines, *The 23rd Annual Computer Security Applications Conference (ACSAC 2007)*, pp.385-397, 2007.

[17] S. Sundareswaran, D. Lin and A. C. Squicciarini, Ensuring distributed accountability for data sharing in the cloud, *IEEE Transactions on Dependable and Secure Computing*, vol.9, no.4, pp.555-567, 2012.

[18] OpenStack.org, *What is OpenStack?*, 2018.

[19] Red Hat, *Digital Transformation, the Open Source Way*, 2020.
[20] OpenStack.org, *Welcome to OpenStack Documentation*, 2019.
[21] OpenStack.org, *OpenStack Installation Tutorial for Ubuntu*, 2017.
[22] OpenStack.org, *Ocata OpenStack Summit Recap*, 2015.
[23] R. Maata, R. Cordova, B. Sudramurthy and A. Halibas, Design and implementation of client-server based application using socket programming in a distributed computing environment, *IEEE International Conference on Computational Intelligence and Computing Research*, pp.1-4, 2017.