SECURE FRAMEWORK FOR VIRTUALIZED SYSTEMS WITH DATA CONFIDENTIALITY PROTECTION

BY

FANGZHOU YAO

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Adviser:

Professor Roy H. Campbell

ABSTRACT

Benefits have been claimed by adopting virtualization techniques in many fields. It could significantly reduce the cost of managing systems, including critical systems used in cyber power grid. However, in such environments, multiple virtual instances run on the same physical machine concurrently, and reliance on logical isolation makes a system vulnerable to attacks. Virtual Machine Introspection techniques show effectiveness in building a more secure virtualized environment, since they simplify the process to acquire evidence for further analysis in this complex system.

However, the VMI technique breaks down the borders of the segregation between multiple tenants, which might lead to the disclosure of cloud tenants' data. This potential threat becomes a concern for virtual instances running critical systems, and hence it should be avoided in a public cloud computing environment. The disclosure of data could happen easily due to compromised connections, both inside and outside of the cloud, and the misuse of the cloud administrator's authorization.

Thus, in this thesis, we focus on building a secure framework, CryptVMI, to address the above concerns. Our approach maintains a client application on the user end to send queries to the cloud, as well as parse the results returned in a standard form. We also have a handler that cooperates with the introspection applications in the cloud infrastructure to process queries and return encrypted results. The introspection application is able to extract information reflecting the behaviors of the guest systems. It also demonstrates its ability to restore processes upon unexpected modification from the remote user.

This work shows our design and implementation of this system, and the benchmark results prove that it does not incur much performance overhead.

To my parents, my grandparents and my uncle, for their love and support.

ACKNOWLEDGMENTS

I am using this opportunity to express my gratitude to everyone who supported me throughout my Master's study in Computer Science.

I am thankful for my advisor, Prof. Roy H. Campbell, for his aspiring guidance and continuing support. I would like to thank Mirko Montanari, Read Sprabery, John Bellessa and Mayank Pundir for their help on my projects. I am sincerely grateful to other members in System Research Group for sharing their truthful and visionary views during the weekly group meetings. Finally, I would like to thank my family and friends for their support and love. My graduate studies have been supported through funding from Trustworthy Cyber Infrastructure for the Power Grid (TCIPG) and Assured Cloud Computing (ACC).

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Situation Awareness

The cloud computing becomes popular its power, as the size of data being stored and processed in industry is increasing drastically. Building and maintaining storage as well as computation infrastructures become trivial in a cloud environment. Some companies build their own private cloud services, and other companies including Netflix [1] and Dropbox [2], tend to use public cloud services, such as Amazon Web Services (AWS) [3]. Virtualization has turned out to be necessary in both private and public cloud computing solutions, because it provides better utilization of resources and reduces the cost by allowing multiple instances of operating systems (OS) owned by multiple tenants to run concurrently on the same physical machine. Benefits have been also shown by adopting virtualization techniques in the cyber power grid [4], such as migrating Supervisory Control and Data Acquisition (SCADA) systems into a virtualized environment. Since VMs share the same hardware resources, the Virtual Machine Monitor (VMM) is able to reduce redundancy for multiple similar SCADA systems. Though multiple OSes are running in their own virtual machines (VM) and sharing the same physical infrastructure, the VMM can still ensure high availability for users [5].

However, sharing the same physical resources brings up security issues. Even though a VM should only be allowed to access its own resources by design, the traffic between VMs essentially breaks down the physical isolation. The logical isolation in cloud environments is built in the software layer and, consequently, the security guarantees are weaker [6]. A compromised VM can easily and quickly spread malware and make the entire system vulnerable to more attacks. Moreover, it takes time for cloud service users to detect those compromises.

While traditional security systems, such as Intrusion Detection Systems (IDS), are able to enforce policies on every node in a network system and detect violations in real-time, the virtualization framework makes cloud environments complex, and hence it is difficult to collect evidence in such systems. These facts lead to the development of Virtual Machine Introspection (VMI) techniques. VMI tools inspect a VM from a trustworthy outside environment. The outside environment usually refers to the host system and it is protected with techniques, such as Trusted Platform Module (TPM) protection, so users are able to access the entire system and acquire the genuine information of a VM to describe the state of the guest system [7]. IDSes can be built with this technique in the cloud for greater attack resistance, while providing an excellent view of the states in VMs [8].

Though using VMI is a simple and convenient approach for cloud users to acquire trustworthy information from their running VM instances, the inflexibility to integrate with existing security monitoring systems and the possibility of exposing confidential information from VMs makes it difficult to apply this technique in the cloud. There are many private cloud infrastructures that have embedded VMI into their systems to enhance security [9], but those VMI frameworks are customized to specific environments, making it difficult to use them in cooperation with existing security monitoring systems, such as Bro IDS [10]. Concerns are also raised for public cloud systems, since the VMI technique might break down the borders of the segregation between multiple tenants [11]. Companies running their services on public cloud infrastructure would not want to expose their application states to the cloud service provider. Moreover, even in a private cloud, the misuse of administrative authorizations might lead to more attacks [12], and hence any opportunities that could be exploited to examine VM states, even by the cloud administrators, should be minimized. This is especially important for a cloud environment running critical systems like SCADA systems for the power grid. Thus, the necessity is obvious to build a secure framework based on VMI for critical systems, which is able to detect attacks effectively and provide self protection functionalities. Such a framework makes the systems running inside observable and easy to manage. The framework should be performant, as many critical systems are used in metering large amount of real-time data [4].

Therefore, we propose CryptVMI, an secure framework built with VMI techniques, to provide cloud tenants complete status of their virtual instances, while keeping confidentiality from possible attackers, including administrators of cloud services. This system also provides users semistructured outputs as results, which could be extended to work with its built-in and other existing security monitoring systems, on the user end to provide better security for their systems, especially when their cloud computing frameworks are running in the public cloud. With the semistructured outputs, users are able to examine the entire states of their VMs from the remote end, even if it is a mobile device. This framework does not require any modification to the network interface of applications connected to guest systems. It also provides an effective strategy to restore processes upon unexpected modification to the VM.

1.2 Thesis Contribution

We design and implement this secure framework based on Virtual Machine Introspection system techniques to provide a simple interface with semistructured data for users to extend this system with other security monitoring systems. We show that CryptVMI keeps confidentiality of users' information from their VMs in the whole encrypted VMI process. We demonstrate that our encryption scheme introduces minimum overhead to system performance.

Furthermore, our built-in policy examiner on the remote end is able to effectively monitor violations happened in a guest system based on user defined policies, and restore modified processes by attacks.

1.3 Thesis Organization

The rest of this work is organized as follows. We first explain our motivation of proposing such a framework with case studies and summarize related works in Chapter 2. Next, we show the big picture of our framework along with its design in Chapter 3 and implementation details in Chapter 4. We then evaluate it through various experiments and analyze the results in Chapter 5. In Chapter 6, we conclude our work and discuss future approaches.

CHAPTER 2

BACKGROUND

2.1 Case Study on Critical System Security

Compromised critical systems could lead to destructive results. SCADA systems are one type of most widely used critical systems in the world. If such systems are modified or turned off in a power plant and grid, power generation and transmission could be suspended due to attacks, and hence the attacks might even destroy the physical power infrastructure and become a serious threat to human life. In this section, we show possible attacks in SCADA systems with real world cases, and also summarize the significant properties that a framework should have in order to guarantee SCADA systems running in a secured environment.

2.1.1 Stuxnet

Stuxnet is the first known worm targeting specific infrastructure like electricity grids, It is very different from other types of worms. Unlike other applications running in an Operating System (OS) with anti-malware software installed, SCADA systems are monitored by Programmable Logic Controllers (PLC). PLCs control critical systems with strict security limits. It is difficult to attack a PLC, because it is usually written in a specific language. However, Stuxnet is able to exploit the vulnerabilities of OSes to target those that construct the PLCs. It injects malicious code into the PLC and hence makes it compromised [13]. Thus, it is able to take control of SCADA systems. The main reason for the successful attack of Stuxnet back in 2010 is that the softwares for PLCs are not well-defined [13]. Thus, we believe that any computer framework that runs, monitors, or creates softwares, could be resilient. The frameworks should be always actively maintained and monitored; It should be also simple enough, as it is shown that more lines of codes of a program would introduce more vulnerabilities.

In addition, Stuxnet could take advantage of the vulnerabilities of OSes. It takes zero-day vulnerabilities to initiate its attack, and those vulnerabilities are usually not noticed until the attack. Moreover, the attacker is able to hide all related malicious processes, which prevents detection of its presence.

Since it is difficult to cover every single corner of an OS to prevent zero-day attacks, there are some zero-day protection techniques were developed based on policy conformance frameworks and they demonstrate the effectiveness in the scope of keeping integrity of a system. Therefore, the monitoring framework for a critical system should be able to provide warning of zero-day attacks. This also requires the monitoring framework to be resilient and trustworthy continuously.

2.1.2 Shamoon

Shamoon is known as the most destructive post-Stuxnet discovery for SCADA systems, which attacked 30,000 workstations and caused the company to spend a week to restore their systems [14]. It corrupts files on a compromised computer and overwrites its Master Boot Record (MBR). During its spreading stage, Shamoon's wiper component embeds or disguises itself into system executables.

Thus, we want to detect the infected processes in a critical system in time, and restore any of them to its secured state if possible. Furthermore, we would like to keep evidence for infected processes for later investigation.

In summary, the framework for a critical system should be able to provide genuine data describing an entire view of the critical system. This fact requires the monitoring framework to be trustworthy all the time. It is also required to restore the system to a clean state upon attacks. Several types of approaches have been discussed. Some research focuses on enhancing security by adding additional security layers into the system, such as creating a middle layer using security-enhanced Linux between SCADA systems and the external network [15], but it makes the whole system complicated with the additional layers and hence incurs a performance overhead. Others manage to acquire evidences to detect attacks, such as the digital forensics framework Forenscope [16], which is effective in obtaining a complete state of the entire system with trustworthy information, but it needs to reset and restore both the hardware and software states.

We have sufficient reasons to build a secure framework to run and monitor critical systems. It should be resilient and simple. The data extracted from this framework should be trustworthy so as to reveal an accurate representation of the monitored parameters. Moreover, it should be able to restore the system to a secure state to neutralize attacks.

2.2 Related Work

Checking the state of a system insides of a cloud environment as a whole is hard. It requires the security monitoring system to be located on the host, but this approach makes the security system susceptible to attacks. Alternatively, if we deploy the monitoring systems in the network, the view will be limited [8]. Livewire takes advantage of VMI techniques to retain the full view of a host-based IDS, but pulls the IDS outside of the host, namely the node in a network, for greater attack resistance [8]. In addition to visibility, an input framework built on top of Bro IDS was presented for flexibility and scalability. It provides a simple yet flexible user interface and support for asynchronous operation, which enables an IDS to analyze high-volume packet streams under soft real-time constraints [17]. Odessa also proposed a solution to build a resilient policy compliance system in the network. It is designed to distribute the evaluation of rules in a scalable fashion, which employs a set of monitoring agents on the nodes in the network and validates global rules in their corresponding verifiers [18]. Thus, we want to collect the evidence from the OS directly, while keeping the security system in a safe state. Furthermore, the evidence obtained should be standardized in a simple form for users or other security monitoring systems, like an IDS. The evidence should be also available for multiple analysis frameworks for stronger resilience.

Nowadays, many users start using cloud services due to the benefits of a flexible and elastic infrastructure. Virtualization techniques and bridged networks are widely used in such environments, which make the system even more complicated. It also leads to more security concerns, such as the weakened isolation and possible misuse of administrative permissions. We can use VMI tools to simplify this complex system, and obtain the entire state of a VM from the safer outside, namely the Dom0 or host for virtual systems running on Xen or KVM, respectively. However, many companies pay for public cloud services to store their data and run computational applications, because of the ease of configuration and maintenance. Thus, the concern for confidentiality becomes inevitable. For instance, if Amazon started providing VMI features for its cloud users, then the administrators in Amazon would be able to access the entire state of every customer's VM running in their public cloud. This is definitely not what Amazon's customers want to happen with respect to their privacy. Furthermore, running critical systems in the public cloud would become an ethical issue due to this possible disclosure of data. Even in a private cloud, the possible misuse of administrative permissions is also a serious security problem. Thus, we want to collect data with simple VMI techniques from VMs through the complex cloud framework, but we also want to keep the confidentiality of any sensitive information obtained during this process.

Therefore, a secure framework in the cloud, which is able to respond to queries from multiple users or security monitoring systems with simple and standardized results would be beneficial. The concern for credibility of data gathered is minimized and the data can be as much as the whole picture of a VM, since VMI work is processed in a trustworthy outside environment. In addition, the queries to VMs and returned results should be encrypted to address the concern of the possible leakage of users' data.

CHAPTER 3

SYSTEM DESIGN

3.1 Threat Model

In a cloud environment, the compute node should not be easily compromised. Hence, systems running VMs are trustworthy environments. This guarantees that the information gathered using VMI is genuine. If the compute node, which is essentially the host system in a virtualized environment, is compromised, it may result in misleading data. However, this should rarely happen. The host systems in the cloud should be protected by TPM, and hence the integrity is usually guaranteed. Nowadays, labeling technique is also widely used, so that all processes and files in a system are labeled in a way that represents security information. Labeling also works in our framework. Though the native support for feature in Linux, like SELinux, is usually disabled for better VMI integration [19], most hypervisors still provide their own schemes for labeling, such as Xen Security Modules (XSM) for Xen [20]. Also, since the host system Dom0 is essentially a VM on Xen hypervisor for management purposes, the security of a host system mainly depends on the VMM. It is obvious that VMM could be also seen as an OS, but it is significantly simpler than standard modern OSes, which has been within 30 thousand lines of code [8]. Moreover, in a mature commercial public cloud service system, cloud service providers also utilize state-of-the-art electronic surveillance and multi-factor access control systems with a 24/7 monitoring service [3]. Thus, a VMM is much more difficult for an attacker to compromise.

We also assume that the VM running in a compute node might be totally malicious. The impact of a malicious VM is still trivial in this case, since we can obtain the entire genuine state of the VM and detect the problem in time. For instance, inside of the malicious VM, a malware might be able to hide itself from a user's inspection, such as the `ps aux` command, but it cannot remove itself from the processes list of the OS running in the VM, otherwise it will not be effective. Even if attackers can exploit a compromised VM and manage to attack the VMM, VMMs like Xen [21] and KVM [22] are able to defend against such attacks with their own security mechanisms. In the worst case, the security of the cloud system will still detect the attack, though it might result in the loss of their data and suspending of their services for cloud users.

Users can access their VMs through a secure connection, such as Secure Shell (SSH), which should be secure enough to avoid most man-in-the-middle attacks if the SSH keys are not obtained by attackers. However, connecting to a compromised VM from the remote end might lead to the infection of users' machines. We expect that users have anti-malware programs on their own OSes.

Furthermore, we presume that co-location attacks [23] might happen. Thus, we do not want to expose the structure of our cloud system to users, such as on which compute node a VM is located, or the IP address of the node. This approach should minimize the chances for such attacks, and it also keeps transparency to users.

Our objective is to prevent potential attackers, including cloud administrators, from knowing the entire state of a user's VM. The disclosure could happen due to the compromised connection between the remote user and the cloud, or the compromised internal connections inside of the cloud as well as the misuse of the cloud administrator's authorization.

In our framework, VMI applications are set up with `root` authorization. We assume that the normal administrators of the public cloud system do not have `root` authorization on compute nodes, since there would be no need for the cloud administrators to gain such authorization, once the cloud has been configured and securely booted with TPM. This assumption is reasonable in reality for a public cloud, as they are in a group with only limited permissions interacting with the cloud service application programming interfaces (API) and VMs, otherwise the administrator can install their own VMI tools, even bypassing the logs in a cloud system. Therefore, configuration files and keys stored with `root` permission for CryptVMI also stay safe. As it is shown in Table 3.1, the access permissions are categorized for various types of contents.

Table 3.1: Access Permissions in CryptVMI Framework

Content for Access	VMI Framework	Admin	User
Running at `root`	Yes	No	No
Cloud Tenant Info	No	All	Only own tenants
VM Location Info	Yes	Yes	No
VMI Keys	All	No	Only own paris
Data in VMs	All	No	Only own VMs

3.2 CryptVMI in the Cloud

In this section, we present concepts that we used to design CryptVMI, as well as the architecture for this VMI system. First, we show show the architecture of this system. Second, we discuss how we can guarantee the integrity of the host system with TPM. Third, we discuss the interface that we used for users' queries and results. Forth, we describe our encryption and decryption scheme. Last, we explain why network transparency for a guest system to its connect external applications or sensors is necessary.

3.2.1 Architecture

The design of CryptVMI has three major components. Figure 3.1 shows the overview of this framework. Dotted lines represent secure connections or connections with encrypted components. There is only one remote user and one compute node shown, but multiple clients and compute nodes are supported in our design. The secure connection from the remote cloud tenants to their owned VMs is not part of the VMI system, but it suggests a general approach that users communicate with their instances.

The first component is the client application along with the policy examiner on the remote user end. It accepts queries from users or other security applications, and directs queries to the handler. It also eventually decrypts results returned from the cloud. The built-in policy examiner extracts system information from the client application and monitors the status of guest systems with user-defined policies. We will explain the methodology that we used for this examiner in next section. The second component handles queries sent from the client application. Once the query is processed, the encrypted data is transferred back to the client. The third component involves strategies to acquire desired results with encryption.

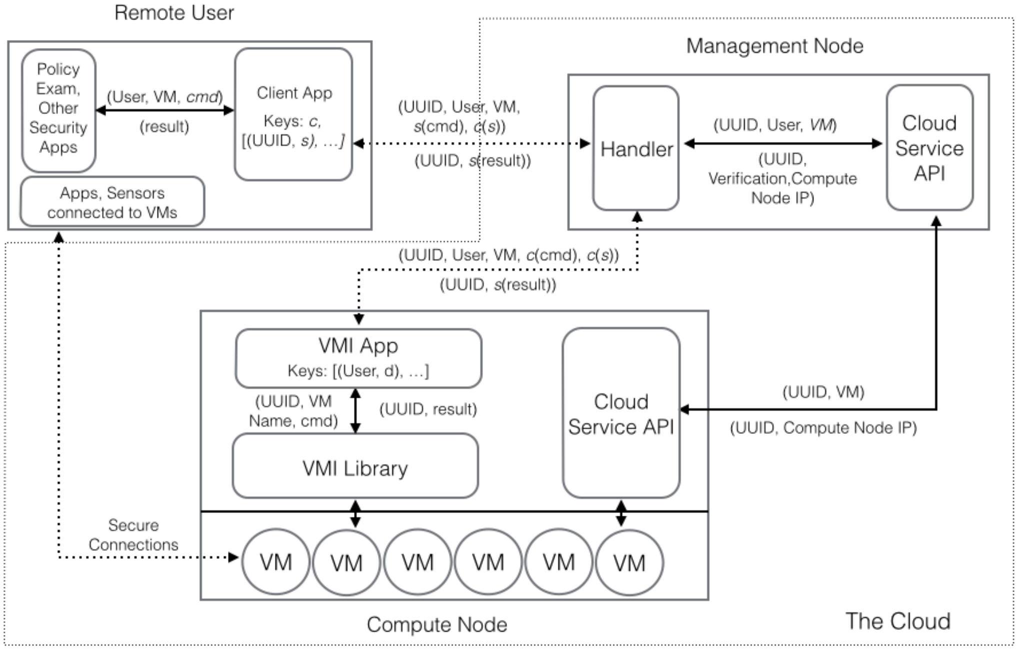

Figure 3.1: A High Level View of the CryptVMI Architecture

User Client

Once a query is accepted, the user client application will assign a unique ID and a random symmetric key s to this query. The command is encrypted with the key s, and the key is encrypted with the user's corresponding public key c. The client then initiates the request to the management node in the cloud environment through a Secure Sockets Layer (SSL) connection, whose address is loaded from the configuration file.

The client application handles the result in its callback function. Once the result is received, the application first decrypts it with key s and then decodes it with Base64, if the query is requesting binary data, such as memory dumps. Eventually, it sends back the data to the query's origination.

Query's origination generally includes the built-in policy examiner deployed on the remote end, however other security monitoring systems are also able to request data from this client in a RESTful fashion. Since the application is written in Ruby and might not be run in `root`, users can modify the code to parse the returned data in different formats, and hence make it easy to cooperate with various applications, such as IDSes.

Query Handler

The request received from the client has five parts: the unique ID of the query, the user's information including the credential token, VM instance name in the cloud system, symmetric key s encrypted with user's public key c and the command encrypted with key s. Once the handler receives this request, it first checks the user's token and the instance name with the cloud service API to verify if there is such a VM associated with this tenant, and if this user has access to the requested instance in the cloud. The handler then uses the cloud service API to locate the IP address of the compute node that holds the designated VM, and obtains this VM's name to the hypervisor. The name of a VM to the hypervisor is usually different from its instance name provided from the cloud service API to the user, and hence this mapping becomes necessary.

Communications in this process are not encrypted, because they are in the internal cloud system network, and the cloud administrator should already know the unencrypted information such as the name of the VM. Since the command and the key s are encrypted, the details will not be disclosed. In addition, the command is essentially a JSON document inside of the request, and hence it is flexible with various parameters. Thus, a new request is sent to the introspection application on the corresponding compute node.

Finally, the query handler sends the encrypted result data back to the client.

Introspection Application

The introspection application manages users' private keys, and hence it decrypts the key s with the the specific user's private key d. Thus, it is able to decrypt the command message. The introspection application translates the message and invokes corresponding VMI tools built on top of the VMI library to acquire desired results.

Results are then encrypted with the symmetric key s and transferred back to the query handler.

Though all three components are written in Ruby, the client application does not have to be run in `root`, since the users usually take control of their own machines and there are only the public keys stored at the their end. Each of these components is described in following subsections. Additionally, there is a series of VMI tools deployed on compute nodes, which are written in C.

3.2.2 Trusted Platform Module

TPM is enabled on both the compute nodes and management nodes in order to assure the integrity of the platform. It guarantees a system to start the boot process from a trusted condition, until the OS has completely booted and applications are running [24].

TPM also contains several Platform Configuration Registers (PCR), which enable the secure storage and is able to report security metrics. The metrics can be used to detect changes in the system. Thus, utilizing TPM is able to assure platform integrity, which provides us a trustworthy host system for our VMI operations on guest systems.

3.2.3 Flexible and Simplified Interface

We want our interface for users or other security monitoring systems to achieve good performance and scalability. Also, the query should be simple and easy to write, while the result is standardized and flexible. Additionally, considering that data transmission through the network is a major part of our system, we want to reduce the network throughput.

RESTful API

Representational State Transfer (REST) is a data style designed for modern web and distributed applications. It ignores the details of component implementation and protocol syntax, in order to provide better performance and scalability of component interactions [25].

Many applications nowadays are using this type of API style, including IDSes [26]. This fact makes our VMI system easily extendable to other security systems, including the built-in policy examiner. We simply accept

HTTP POST requests in an asynchronous fashion.

Query and Result in JSON

We use Javascript Object Notation (JSON) for queries and results. Its semistructured schema makes JSON flexible for various types of data, since it does not have a regular structure [27]. Attributes in a JSON document are self-described, and hence it is clear for users to understand and easy to write. It also simplifies the structure of the document, compared to XML. Thus, using JSON reduces the throughput when transmitting data over the network.

A simple example of query in JSON from users or other security applications would be like below.

```
{
  "user": "[token]",
  "vm": "[vm_name]",
  "command": { ... }
}
```

The result is similar, but binary data, such as the memory dump, is encoded with Base64 [28].

3.2.4 Encryption and Decryption

Since the query itself might disclose the information [29], such as the `pid`, we should encrypt both the query and result. We use a generated symmetric key as the session key to encrypt and decrypt parts of the query and the result. Moreover, each user has a unique pair of public and private keys. We use the public key on the user end to encrypt the symmetric key, while decrypting it with the private key stored in the VMI application end. Each query session is associated with a unique random symmetric key. Figure 3.2 shows our encryption scheme.

We use this scheme for two reasons. One reason is that public key encryption, namely the asymmetric key encryption, does not support encryption for large data by itself [30]. Thus, this scheme allows us to encrypt large data,

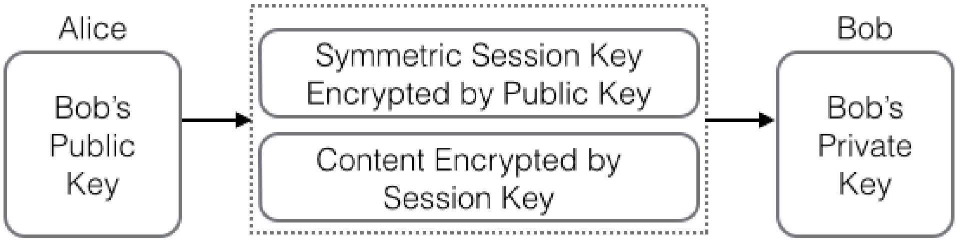
Alice
Bob's Public Key
Symmetric Session Key Encrypted by Public Key
Content Encrypted by Session Key
Bob
Bob's Private Key

Figure 3.2: Encryption Scheme Used in CryptVMI

such as a 1GB memory dump, and achieve the nearly equivalent security level as the asymmetric key encryption. CryptDB uses similar approach to allow for query execution on an encrypted datastore [29]. The other reason is for performance. RSA is not as performant as symmetric encryption schemes [30]. Additionally, modern Intel processors have embedded AES symmetric key encryption hardware accelerators, and applications can benefit greatly from using AES on these processors [31].

3.2.5 Network Transparency

In our framework, the guest systems is connected to the external network for their general purposes, such as a SCADA applications connect to external power grid applications and sensors. However, the remote end running those applications and sensors should be able to access the SCADA systems running in guest systems, namely DomUs, and hence the VM inside of this framework should provide an interface for connections from the outside. Moreover, the remote end users should not discover that the SCADA system is in a VM and they should be able to access it as usual. Thus, we need to connect the two segments of networks together. We use the bridge scheme provided by the VMM and OS on the host system, which reconfigures its physical network into a bridged network. Moreover, adopting this approach could reduce the chance for co-location attacks, which make it possible for attackers to attack co-located VMs from a compromised VM, once they detect that the compromised system is in a VM.

3.3 Policy Examiner at the Remote End

In this section, we describe our design for the policy examiner deployed on the remote end. In Figure 3.1, we show that both the policy examiner and other applications and sensors for general purposes are on the same node. However, in the real world, they do not necessarily have to be on the same machine.

3.3.1 Architecture

The design of the Policy Examiner has two major components. Figure 3.3 shows the overview of this system.

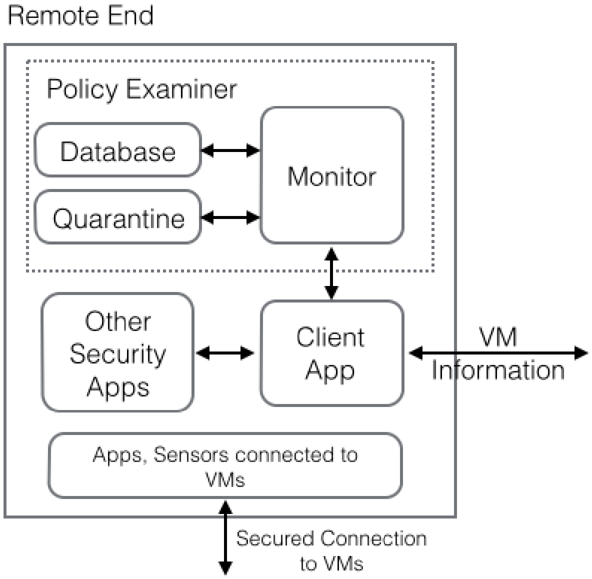

Figure 3.3: The Architecture of the Remote End

The first component is the monitor. It calls the client application to examine states of DomUs and keeps monitoring the VM with policies defined in the database in a certain time period. It also records critical events into database when a violation is detected. The second component is the database, which stores policies and records the critical events.

3.3.2 Policy-Based Monitoring

We use VMI techniques to discover the complete state of a VM, since the genuine and trustworthy data extracted from a VM with VMI is able to reflect the state of a VM faithfully. Also, the integrity of the host systems, namely the Dom0s, are protected by TPM. The process and module lists are two major facts acquired to analyze the critical systems with our policy examiner, as they are useful to detect hidden processes and modules, which might be malicious and cannot be detected by introspection from the inside of a VM.

Processes are one of the most important part in a critical system, such as SCADA applications in SCADA systems as they monitor the data coming from the power grid. In order to verify the integrity of such applications, we focus on their code segments. The code segment of a process is one of its sections in memory that contains executable instructions. Its size is fixed and it is usually read-only within the scope of the guest system, though we are able to modify it from the host system. Thus, a significant characteristic of an infected program is the change of its code segment. Based on this fact, we can verify the integrity of a process by comparing the hash value of the memory dump of its code segment with the one dumped in a clean state. The dump of the process from a clean state could be handled from a clean image template, or other systems running with the same OS or applications that are guaranteed to be secure. If we detect that the hash value is different from what we have on file for a process, we can restore the process to its clean state by sending encrypted commands and clean dumps to the cloud. The VMI application is then able to overwrite the code segment with our clean dump.

We enforce our security policies in a white list fashion. Users are able to define their own policies for entities like processes and modules, and keep them in the database on the host system. The policy model for each tuple in the table consists of attributes like module and process names, permitted users and running time intervals, hash values of the code segments and paths to their binary dumps processed in a clean state of the VM. The policy examiner monitors VMs with defined policies in a pre-set time period. Any violations, such as process names appearing in the process list but not in the white list, will be reported and recored. Moreover, the tampered process will be detected by the hash value of its code segment. In addition, since we want to obtain consistent snapshots from a VM, we pause the VM while gathering its process and module list. However, since the code segment is read-only to the guest system, dumping code segments does not necessarily need a pause.

3.3.3 Evidence Gathering

In order to reproduce the problems of the system, or any attacks, we record the critical events with a time stamp into the database upon any violations. We take a sample of the unexpected modification before restoring it to its original safe state. For instance, we would dump the code segment of the infected process and store it, before we restore the process to its clean state. We only keep the path as part of the record tuple in the database, which leads us to the infected dump in the quarantine.

CHAPTER 4

IMPLEMENTATION DETAILS

4.1 Simulation to the Public Cloud

Systems were set up with a 64-bit CentOS Linux for reliability and simplicity, since it is an actively maintained Linux variant and its code base is relatively small. Hash values of processes from the guest systems, were calculated in a clean VM state with MD5 algorithm. They are stored in a MySQL database on the remote end.

We set up our experiments with OpenStack to simulate a public cloud environment. We want to simulate a public cloud in real life, such as Amazon Web Services (AWS), which uses Xen hypervisor and runs VMs in the para-virtualization mode [3]. However, the VMs in our cloud system for tenants were running with full-virtualization enabled with the Xen hypervisor in order to support unmodified OSes, such as different types of SCADA systems running on various OSes.

4.2 Cloud API

OpenStack is a popular solution for hosting private clouds developed by Rackspace and NASA [32]. It has various components addressing necessities of a cloud system. We deployed the components needed by the controller node and network node on the same machine, and made it our management node. Even though the compute node can be the same machine as the management node, in order to simulate a public cloud environment, we used a separate machine with the same hardware configuration as our controller node. If there are multiple compute nodes in this cloud system, each of them should have its own running copy of the introspection application.

The OpenStack APIs are also designed in a RESTful manner, and hence our CryptVMI applications can communicate with them easily using industry standard libraries.

4.3 CryptVMI

We implemented the three major CryptVMI components in Ruby for three reasons. The first reason is its natural affinity with JSON documents, because the dictionary data structure in Ruby can be easily converted into JSON with the Ruby gem `json`. The second reason is that it provides gems, namely the libraries for Ruby, to help us for faster development. We used Ruby Encrypt in our symmetric key encryption, which simplified the process and hence we could only keep a 32-byte random string as the key for the AES-CBC-256-bit encryption [33]. The third reason is that Ruby is an actively maintained language, therefore security issues can be fixed in time.

4.4 Virtual Machine Monitor

We used the Xen hypervisor to simulate AWS [3]. It has been usually considered that Xen is not as secure as KVM [34]. KVM can be less dependent than Xen, because it is essentially a kernel module [22], while Xen needs to modify the OS kernel [21]. However, Xen is a better choice in our framework. As it is mentioned in previous chapter and Figure 4.1, the security for Xen virtualization mainly depends on the bare metal hypervisor instead of Dom0. Dom0 functions similarly to the host system in KVM, but it is a para-virtualized VM on Xen hypervisor. In KVM, the host system is the Linux system and it is obvious that the Linux OS is implemented with much more lines of code than Xen hypervisor, which might lead to more uncertainties. Furthermore, the VMI library that we utilized requires a patch to KVM [19], and hence it will make the VMM possibly unreliable, which is what we want to avoid.

Therefore, we still chose to use Xen as our VMM.

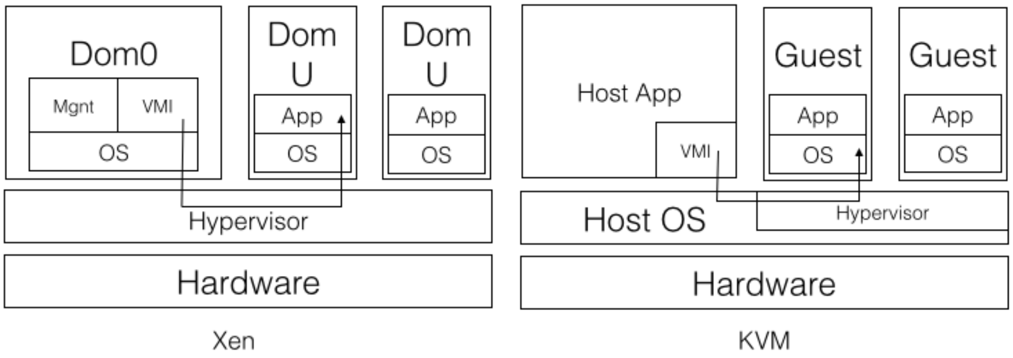

Figure 4.1: Virtual Machine Introspection in Xen and KVM

4.5 Virtual Machine Introspection Library

We built our VMI application on top of LibVMI, which inherits features from XenAccess. XenAccess is designed based on the concepts of virtual memory introspection and virtual disk monitoring. Consequently, its APIs allow applications built on top of it to access the memory and disk of a specific virtual system [35]. It does not require any modification to the Xen hypervisor, and hence it has small performance and reliability impact. LibVMI also provides functions for accessing CPU registers, pausing a VM and printing binary data [19].

Maitland introduces a lightweight VMI system designed specifically to detect malware through packer detection [36]. The system achieves good performance, but it does not allow user to customize their queries. Furthermore, it is limited to para-virtualization environments.

By building CryptVMI on top of LibVMI, we can have supports for both fully and para-virtualized environments, and we are able to write introspection tools for different payloads.

4.6 Virtual Machine Introspection Use Cases

In order to obtain meaningful data from a VM, it is essential to know the offsets for processes in `task_struct`. In order to obtain these offsets, we need to create a kernel module in a VM with a clean state, which should not be malicious and is usually the initial state of an image template. This is because each version and variant of OSes has various memory offsets [37]. The code below illustrates how to obtain these offsets for VMI applications, such as those for `pid`, `mm` and `mm_struct`.

```
struct task_struct *p = current;
pid_offset = (unsigned long)(&(p->pid))
    - (unsigned long)(p);
mm_offset = (unsigned long)(&(p->mm))
    - (unsigned long)(p);
codeseg_start_offset = (unsigned long)(&(p->mm->start_code))
    - (unsigned long)(p->mm);
codeseg_end_offset = (unsigned long)(&(p->mm->end_code))
    - (unsigned long)(p->mm);
```

A two-layer translation to the target address inside of `mm_struct` could be processed, with required offsets obtained. Once the process with the designated `pid` is found in the iteration of the process list, the VMI application could obtain the address of `mm_struct` in its `task_struct`. The code below shows then the VMI application is able to access the `mm_struct` to acquire the target's virtual memory addresses, which could be, but not limited to, the starting and ending addresses of the code segment for a process. In addition, we can then calculate the size of the code segment using the two addresses, which is essentially $1+|diff_{codeseg}|$, where $diff_{codeseg}$ is the difference between the two addresses.

```
vmi_read_32_va(*vmi, next_process + mm_offset - tasks_offset,
    0, (uint32_t*) &mm_addr);
vmi_read_32_va(*vmi, mm_addr + target_offset,
    0, (uint32_t*) &result_addr);
```

As it is shown in the code below, the VMI application uses the virtual addresses to read the segments and then write them to files. This uses the LibVMI's built-in translation from the virtual memory address to the physical address.

```
vmi_pause_vm(*vmi);
vmi_read_va(*vmi, start_codeseg_vaddr,
    pid, buffer, code_size);
```

```
fwrite(buffer, 1, code_size, code_file);
vmi_resume_vm(*vmi);
```

The code below shows how we are able to restore the code segment from a clean dump.

```
addr_t curr_vaddr = startcode_vaddr;
while (!feof(original_code)) {
         fread(file_buffer, 1, 1, original_code);
         vmi_read_8_va (*vmi,
            curr_vaddr, pid, mem_buffer);
        if(*file_buffer != *mem_buffer) {
            vmi_write_8_va(*vmi, curr_vaddr,
                pid, file_buffer);
}
        curr_vaddr ++;
}
```

Even though the code segment field is read-only, a process might be terminated during the dumping process. In order to acquire consistent memory reading, the guest system is paused during the memory access.

CHAPTER 5

EVALUATION

In our experiments, the user's client machine, the management node and the compute node have the same hardware configuration with an Intel Core i7 CPU at 3.40 GHz. The DomU in this experiment, namely the guest system, is allocated with 1 vCPU and 1GB memory. The same sets of experiments were conducted 10 times, so the time measurements below are averages.

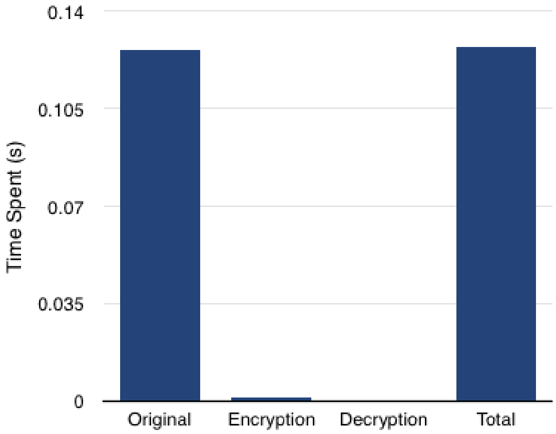

Figure 5.1: Time Distribution to Extract the Processes List from a VM

Since we want to ignore the time consumed during the network transmission of data, we benchmarked the code used in CryptVMI for the encryption and decryption of data. Thus, the total time was the estimated time for CryptVMI to finish the VMI task without considering network transmissions, while the original time was the time consumed in finishing traditional VMI tasks. Figure 5.1 shows the results of the average time by running VMI acquisition for the processes list from a DomU. The overhead introduced is very little and barely noticed.

Figure 5.2 shows the time spent in dumping and hashing the code segments of the 1.4KB experiment process, compared to the time for dumping code

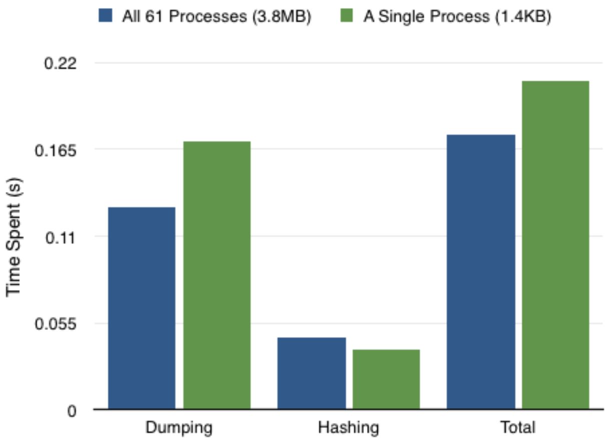

Figure 5.2: Time Distribution to Dump and Hash Code Segments

segments from all 61 processes inside of the VM. The total time is the sum of the time spent in dumping and hashing processes. We can see that the two values are very close. The reason could be two-fold. One is that our VMI application essentially iterates the complete process list it is instructed to dump only one process. The other is that the loading time of the VMI libraries might take a large part of the time spent. Thus, we should be able to improve our results by caching the starting and ending addresses of code segments in a hash map, instead of iterating the primitive Linux `task_struct` list, namely the process list. However, this works only if processes in the VM do not change often. Furthermore, if we look at the time spent in calculating hash values, respectively, they are also similar. This is because the loading time of the Ruby libraries plays a major role. Though it might take around 0.16s to dump the code segment of a process, since the VMM does not pause the VM, it should not cause any starvation of tasks from the VMM and the remote end.

Figure 5.3 shows the time spent in restoring the experiment process. It involves two steps, hashing to determine if there is a need to restore and writing from the host system to the process. Thus, the total restoration time spent includes the time spent in both hashing and restoring the code segment from the host system disk back to the guest virtual instance. As it can be observed, the time is equivalent to dumping the code segment. We think that the I/O operation takes a significant portion of it, but it is still very performant as they can be done in around 0.18s.

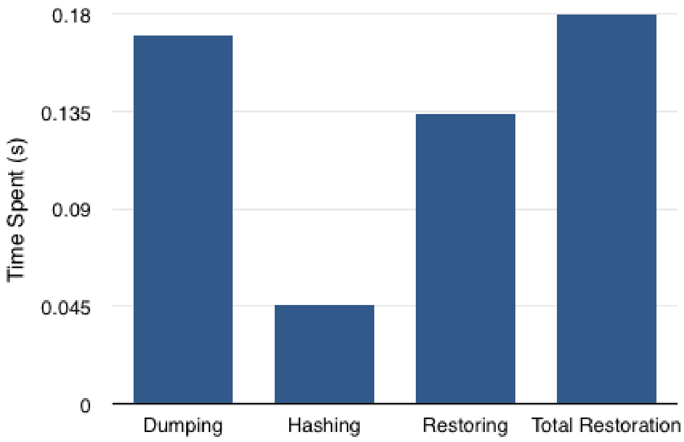

Figure 5.3: Time Spent in Restoration

In Figure 5.4, we can see the results of dumping the whole 1GB memory from a DomU. The encryption and decryption roughly took the same time as dumping the memory, which was about 10 seconds. We label the total time for dumping the 1GB memory as the CryptVMI processing time in Figure 5.5. In this way, we can compare the CryptVMI dumping with encryption time with the time consumed in transferring the dumped data in the cloud system and between the remote client and the management node, respectively. The results show that the CryptVMI processing time is only a small portion, compared with the time consumed during transmitting data through networks.

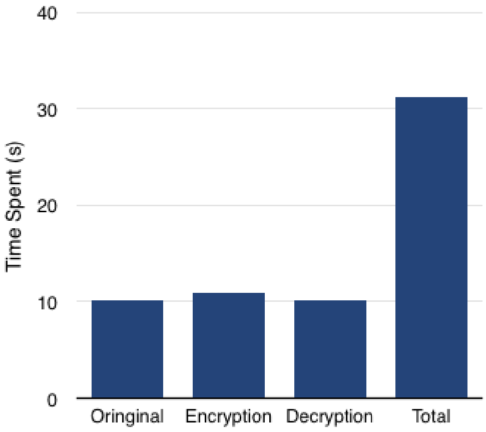

Figure 5.4: Time Distribution to Dump 1GB memory from a VM

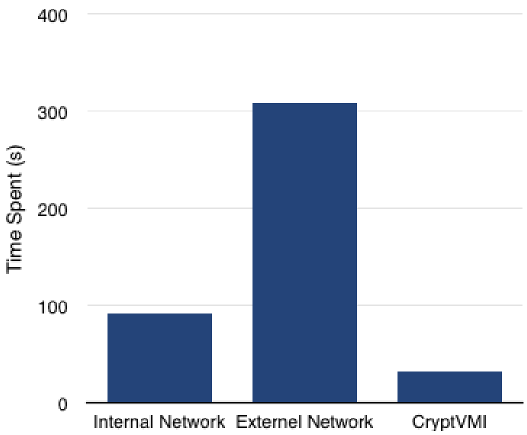

Figure 5.5: Time Consumed in Data Transmission in Networks and in CryptVMI

Though the overhead is not much in total, the main reason for this overhead can be concluded as the slow I/O operations on the disk. Our introspection application calls the VMI tools to dump the memory. The dump is written into a file by the VMI tools and then we read the file into memory for Base64 encoding and encryption. Once the data is sent to the user client application, the user client decrypts it and saves the encrypted dump into a temporary file. The client then decodes it for users and other applications.

CHAPTER 6

CONCLUSION AND FUTURE WORK

This work is based on the author's past published papers [38, 39, 40]. In this thesis, we show our design and implementation of CryptVMI, a secure framework based on Virtual Machine Introspection, to provide flexibility in its interface to cooperate with other security monitoring systems while keeping the confidentiality of users' data, especially in the public cloud. It is able to provide trusted information regarding the complete states of systems, while keeping network transparency to the connections from the outside. The policy examiner monitors the critical systems with user defined policies, and gathers evidence upon violations. Our benchmarks show that CryptVMI only introduces little overhead in performance. We believe that this solution is able to address the concerns of the multi-tenancy public cloud, while also providing an interface for enhanced security.

In the future, we plan to integrate this encryption feature into LibVMI by modifying the library, instead of relying on application level encryption schemes. This approach should also increase the performance of our system as dumps would be then encoded and encrypted in the memory. Eventually, we want to integrate with more security monitoring systems, such as IDSes, through our flexible RESTful APIs.

REFERENCES

[1] Amazon Web Services, "AWS Case Study: Netflix, [Online]," Available: http://goo.gl/jrVvI0.

[2] Dropbox, "Where Does Dropbox Store Everyone's Data?, [Online]," Available: https://www.dropbox.com/help/7/en.

[3] Amazon Web Services, "Amazon Web Services, [Online]," Available: http://aws.amazon.com/.

[4] K. Wilhoit, "SCADA in the Cloud: A Security Conundrum?" *Trend Micro Incorporated Research Paper*, 2013.

[5] T. Burger, "The Advantages of Using Virtualization Technology in the Enterprise, [Online]," Available: http://goo.gl/2oqZgo.

[6] M. Factor, D. Hadas, A. Hamama, N. Har'el, E. K. Kolodner, A. Kurmus, A. Shulman-Peleg, and A. Sorniotti, "Secure Logical Isolation for Multi-tenancy in Cloud Storage," in *Proceedings of 30th IEEE International Conference on Massive Storage Systems and Technology*, 2013.

[7] K. Nance, B. Hay, and M. Bishop, "Virtual Machine Introspection: Observation or Interference?" *IEEE Security and Privacy*, 2008.

[8] T. Garfinkel and M. Rosenblum, "A Virtual Machine Introspection Based Architecture for Intrusion Detection," in *Proceedings of 10th Annual Network and Distributed System Security Symposium*, 2003.

[9] J. Konstantas, "VM Introspection: Know Your Virtual Environment Inside and Out, [Online]," Available: http://goo.gl/Yd4ioQ.

[10] The Bro Project, "The Bro Network Security Monitor, [Online]," Available: https://www.bro.org/.

[11] C. Brenton, "Introspection: Boon or Bane of Multitenant Security?, [Online]," Available: http://goo.gl/5aIj6W.

[12] S. Butt, H. A. Lagar-Cavilla, A. Srivastava, and V. Ganapathy, "Self-Service Cloud Computing," in *Proceedings of 19th ACM Conference on Computer and Communications Security*, 2012.

[13] P. Marks, “Why the Stuxnet Worm is Like Nothing Seen Before, [Online],” Available: http://goo.gl/KHhvhU.

[14] H. MacKenzie, “Shamoon Malware and SCADA Security? What are the Impacts?, [Online],” Available: http://goo.gl/V0fSUp.

[15] R. Bradetich and P. Oman, “Connecting SCADA Systems to Corporate IT Networks Using Security-Enhanced Linux,” in *Proceedings of 34th Annual Western Protective Relay Conference*, 2007.

[16] E. Chan, A. Chaugule, K. Larson, and R. H. Campbell, “Performing Live Forensics on Insider Attacks,” in *Proceedings of 2010 CAE Workshop on Insider Threat*, 2010.

[17] B. Amann, R. Sommer, S. Aashish, and S. Hall, “A Lone Wolf No More: Supporting Network Intrusion Detection with Real-Time Intelligence,” in *Proceedings of 15th International Conference on Research in Attacks, Intrusions, and Defenses*, 2012.

[18] M. Montanari, E. Chan, K. Larson, W. Yoo, and R. H. Campbell, “Secure and Flexible Monitoring of Virtual Machines,” in *Proceedings of 26th IFIP International Information Security Conference*, 2011.

[19] B. D. Payne, “Simplifying Virtual Machine Introspection Using LibVMI,” *Sandia Report*, 2012.

[20] X. Project, “Xen Security Modules: XSM-FLASK, [Online],” Available: http://goo.gl/9WlBQ8.

[21] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, “Xen and the Art of Virtualization,” *ACM Special Interest Group on Operating Systems Review*, vol. 37, no. 5, 2003.

[22] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, “KVM: the Linux Virtual Machine Monitor,” *Linux Symposium*, vol. 1, 2007.

[23] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, “Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-party Compute Clouds,” in *Proceedings of 16th ACM Conference on Computer and Communications Security*, 2009.

[24] S. Bajikar, “Trusted Platform Module (TPM) based Security on Notebook PCs White Paper, [Online],” Available: http://goo.gl/Ya57Wv.

[25] R. T. Fielding and R. N. Taylor, “Principled Design of the Modern Web Architecture,” *ACM Transactions on Internet Technology*, vol. 2, no. 2, 2002.

[26] M. Rouached and H. Sallay, "RESTful Web Services for High Speed Intrusion Detection Systems," in *Proceedings of 20th IEEE International Conference on Web Services*, 2013.

[27] D. Crockford, "The Application/JSON Media Type for JavaScript Object Notation, [Online]," Available: http://tools.ietf.org/html/rfc4627.

[28] S. Josefsson, "The Base16, Base32, and Base64 Data Encodings, [Online]," Available: https://tools.ietf.org/html/rfc4648.

[29] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan, "CryptDB: Protecting Confidentiality with Encrypted Query Processing," in *Proceedings of 23rd ACM Symposium on Operating Systems Principles*, 2011.

[30] L. M. Adleman, R. L. Rivest, and A. Shamir, "Cryptographic Communications System and Method," *U.S. Patent No. 4,405,829*, 1983.

[31] S. Gueron, "Intel Advanced Encryption Standard Instructions Set, [Online]," Available: hhttp://goo.gl/2Zp3OW.

[32] Open Stack, "Open Source Software for Building Private and Public Clouds, [Online]," Available: https://www.openstack.org/.

[33] J. Dwyer, "Decrypting Ruby AES Encryption, [Online]," Available: http://goo.gl/THA4Sa.

[34] D. Perez-Botero, J. Szefer, and R. B. Lee, "Characterizing Hypervisor Vulnerabilities in Cloud Computing Servers," in *Proceedings of ACM International Workshop on Security in Cloud Computing*, 2013.

[35] B. D. Payne, M. D. P. de A. Carbone, and W. Lee, "Secure and Flexible Monitoring of Virtual Machines," in *Proceedings of 23rd Annual Computer Security Applications Conference*, 2007.

[36] C. Benninger, S. W. Neville, Y. O. Yazir, C. Matthews, and Y. Coady, "Maitland: Lighter-Weight VM Introspection to Support Cyber-Security in the Cloud," in *Proceedings of 5th IEEE International Conference on Cloud Computing*, 2012.

[37] R. Love, "Linux Kernel Development 3rd Edition," *Addison-Wesley*, 2010.

[38] F. Yao and R. H. Campbell, "CryptVMI: Encrypted Virtual Machine Introspection in the Cloud," in *Proceedings of 7th IEEE International Conference on Cloud Computing*, 2014.

[39] F. Yao, R. Sprabery, and R. H. Campbell, "CryptVMI: A Flexible and Encrypted Virtual Machine Introspection System in the Cloud," in *Proceedings of 2nd ACM International Workshop on Security in Cloud Computing*, 2014.

[40] F. Yao and R. H. Campbell, "SafeBox: SCADA Systems in a Secure Framework,," in *Proceedings of 5th Analytic Virtual Integration of Cyber-Physical Systems Workshop*, 2014.