NODEGUARD: A Virtualized Introspection Security Approach for the Modern Cloud Data Center

Maha Shamseddine[1], Auday Al-Dulaimy[2], Wassim Itani[3], Thomas Nolte[2], Alessandro V. Papadopoulos[2]
[1]*Texas A&M University - Corpus Christi*, USA, maha.shamseddine@tamucc.edu
[2]*Mälardalen University*, Sweden, {name.surname}@mdu.se
[3]*University of Houston-Victoria*, USA, itaniw@uhv.edu

2022 22nd International Symposium on Cluster, Cloud and Internet Computing (CCGrid) | 978-1-6654-9956-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/CCGRID54584.2022.00093

***Abstract*—This paper presents NODEGUARD, a security approach for detecting and isolating misbehaving Virtual Machines (VMs) in multi-tenant virtualized cloud data centers, based on the Virtual Machine Introspection (VMI) monitoring primitives. NODEGUARD employs a divide-and-conquer strategy that checks logical groups of VMs to ensure the efficiency of the detection mechanisms which opportunistically approaches a complexity of $\mathcal{O}(\log_2(n))$ when there is a relatively low number of hostile VMs. This greatly enhances the algorithmic time complexity of the proposed security system compared to the $\mathcal{O}(n)$ complexity achieved by the traditional VMI inspection strategy that checks each VM separately. The approach has been evaluated in a virtualized cloud environment using the Mininet network emulator.**

***Index Terms*—Cloud computing, Security, Virtual machine introspection, VMI, Intrusion detection, Time complexity.**

I. INTRODUCTION

The rapid growth of the cloud computing paradigm is accompanied by security concerns that need to be investigated and managed [1]. VMs can be considered as the most vulnerable part of the virtualized system of a cloud environment, as they are easily accessible by tenant users. Thus, there is a need for a security model to protect VMs from any possible attack. VMI is a technique for monitoring the runtime state of VMs and can be used for several security objectives such as malware and intrusion detection. Although its benefit when it comes to security, VMI has a primary drawback which is the performance [2]. Conventional VMI techniques have a high overhead [3]. Moreover, existing solutions that attempt to bring VMI to cloud computing environments add significant overhead to the monitored processes running inside VMs [4].

In this paper, we propose the design and implementation of a VMI-based security approach for detecting malicious processing nodes in virtualized cloud data centers. The proposed approach, called NODEGUARD, aims to improve the algorithmic time complexity to detect the misbehaved VMs. This is done by employing a divide-and-conquer strategy to reduce the search space of the running VMs.

NODEGUARD leverages the VMI interposition and introspection functions to detect, locate, and isolate any source of maliciousness in the tenants' VMs. The approach relies on a set of distributed task probing mechanisms to recursively locate misbehaving VM nodes in the data center. The VMI initiates the execution of a preconfigured collection of distributed algorithm subtasks on the VMs of each cloud tenant and checks the accumulative final result for correctness in the corresponding SLA-specified time frame. Once an erroneous result or an irrational delay in execution is detected on a probed set of tenant's VMs, the algorithm marks the group as malicious, and recursively divides it into two groups, repeating the probing and divisions on each group that further produces malicious results. Therefore, recursive probing continues with the divisions that trigger an error and discards the divisions that operate correctly. This will continue until the misbehaving VM (or group of VMs) is located. The periodic probing mechanisms in the data center result in timely detection and reporting of misbehaving nodes. This results in a relatively low number of hostile VM and, hence, renders the algorithmic complexity of locating a malicious VM node opportunistically approach $\mathcal{O}(\log_2(n))$. Only the VMs detected as misbehaving by the probing algorithm are deeply inspected by the VMI mechanisms to identify the specifics of the source of misbehaving rather than checking each VM individually as is done in traditional VMI intrusion detection approaches.

The rest of this paper is organized as follows. Section 2 presents a background on the VMI concept. Section 3 discusses the existing work focused on intrusion detection to secure VMs using VMI. Section 4 presents the threat model of the attacker that may corrupt the processing logic executing VMs. The proposed solution to detect misbehaved VMs in an improved time complexity and the performance evaluation of the solution are presented in Section 5 and Section 6 respectively. Finally, we conclude the paper with a conclusion in Section 7, which also highlights the future directions of this work.

II. BACKGROUND

Virtualization is the core technology in cloud computing. It is organized by a hypervisor (also called Virtual Machine Manager (VMM)). VMMs virtualize all hardware resources, allowing multiple VMs to transparently multiplex the resources of the Physical Machine (PM). Thus, VM can be defined as software that emulates the properties of a separated PM. It appears as another program running on the host operating system [5]. VMM is the mechanism that facilitates the construction of (VMI). As stated in [6], VMI can leverage three main properties of VMMs, they are: (i) *Isolation*: to

This work has been performed with the support from the Swedish Knowledge Foundation (KKS) under the SACSys project, and from the Swedish Research Council (VR), under the PSI project.

978-1-6654-9956-9/22/$31.00 ©2022 European Union
DOI 10.1109/CCGrid54584.2022.00093

ensure that any software running inside a VM cannot access software running inside another VM or in the VMM, (ii) *Inspection*: It means the VMM ability to access a VM state which includes CPU state (e.g., registers), memory, and I/O device state (e.g., the contents of storage devices and register state of I/O controllers), and (iii) *Interposition*: It means that VMM can interpose on certain VM operations, like executing privileged instructions. Leveraging these properties is essential for the proposed algorithm. The original VMI architecture was presented in [6] for Intrusion Detection. In that architecture, VMM can provide an interface for communication with the VMI Intrusion Detection System (VMI IDS). The VMI IDS communicates with the VMM via commands over this interface. There are three types of commands: *Inspection commands* to examine a VM state, *Monitor commands* to know when certain events occur and request notification through an event delivery mechanism, and *Administrative commands* to control the execution of a VM.

III. Related works

The existing related works are presented and discussed in this section.

As the VMI approach has high overhead [3], [2], many works tried to minimize such overhead. The work in [7] tried to minimize the performance overhead by integrating some of the VMI operations into the hypervisor. The authors in [8] leveraged the most commonly used VMI techniques to monitor VM's status by executing VMI analysis scripts in the hypervisor domain. However, these works only focused on the performance without considering the security issues.

There exist some works focused on intrusion detection for securing VMs using VMI. In [6] the authors presented the VMI approach that utilizes the host-based IDS and expands it outside of the host for maximizing attacks' resistance. VMI mainly depended on the VMM capabilities and endeavored to leverage these capabilities to completely mediate interactions between the host software and the underlying hardware. The work in [9] employed VMI and Machine Learning techniques at the VMM for presenting a security architecture, called *VMGuard* aiming at detecting hidden malware by performing memory introspection. The work in [10] trained a machine learning-based classification approach for detecting malware in a cloud computing environment. The approach optimized a balance between the performance of malware detection and the overhead of the VMI-based system. In [11], an integrated security architecture was presented. The architecture utilized some features from the VMs, such as the resources dedicated to VMs, types of the applications, and policies associated with groups of VMs corresponding to a distributed application, to secure VMs themselves. In addition to VMs' features, the architecture relied on exchanging information among various security components, such as policy-based access control and intrusion detection techniques to detect changing attacks. In [12] presented a VMI monitor approach, called T-VMI, to ensure the security of a specific VM in a certain host. T-VMI is aimed at avoiding the malicious subversion of the routine of VMI and maintaining the integrity of VMI code using some VMI features such as isolation and correctness. The authors in [13] utilizing VMI in proposing an approach to protect VMs by monitoring them and recording their events. The approach defined a policy engine that works in two phases: an offline training phase to collect the accepted processes from trusted VMs, and an online run-time phase to decide on the valid actions. In [14], a VMI-based security framework architecture is presented. It models the behavior of processes running on VMs and uses the multi-threading capability to analyze VMs' activities and control all their events. In [15], the authors integrated the concept of VMI with the *Drakvuf* [16] (which is a dynamic malware analysis system) to extract behavioral characteristics of malware, aiming at protecting the systems from any possible attacks. In [17], the authors proposed an introspection approach, called *VMShield*, for detecting malware and securing virtual nodes in cloud computing environments. *VMShield* monitors the behavior of processes in the run-time by performing virtual memory introspection from the hypervisor. However, all the mentioned works did not consider the complexity of their approaches.

In our previous works [18], [19], we presented an approach to detect malicious nodes in the SDN data plane and categorizing any present attacks by utilizing network programming and probabilistic sketching. However, these works did not consider the data processing in VMs, but they inspired the current work in terms of time complexity.

IV. Threat model

The threat model assumed in this work is represented by an attacker model that has the full capability to initiate modification attacks on the tenants' VMs in the data center and corrupt any processing logic executing on these VMs. The attack vector could target the application services and shared libraries and APIs running on the VM or the guest VM OS itself. In such a scenario, the attack could leverage the malicious services of a virus, trojan horse, or any form of malware that can disrupt the normal operation of the processing software, the supporting libraries, and interfaces, or the underlying OS. Moreover, the attacker is capable of executing controlled denial of service (DoS) attacks on the tenants' VMs which may result in unacceptable processing delays beyond what is tolerated by the specifications of the SLA. The probing logic as well as the interpretation of the probing output reside in the VMI module which is assumed safe and isolated from the cloud physical hosts and, hence, from attacker tampering. The VMI code space is technically very minimal compared to the code size of the VM guest OS. This fact supports the feasibility of ensuring the safety of the VMI module and its invulnerability to malicious code and software bugs which we believe justifies the above assumption. The NodeGuard model is designed to periodically initiate a set of well-specified processing probing tasks on the tenants' VMs and leverages VMI introspection and interposition for ensuring the correct operation of the VM services. This ensures the detection of an attacker tampering with the state of the

VM, its OS, or its running services. Moreover, the model can detect any irrational VM processing time beyond an acceptable threshold specified by the SLA. The main property that allows the proposed security algorithm to succeed in detecting malicious/misbehaving VMs is represented in the design and utilization of transparent probing mechanisms that are viewed as any normal processing tasks to the tenant VMs. This is why, as will be shown later in Section 5, the task selection is done randomly from a deliberate pool of distributed algorithms with pre-configured input and output parameters. Once any source of misbehavior is detected and isolated among the processing VMs, the respective nodes are deeply inspected by the VMI system to detect the cause of misbehavior and whether it is benign due to a misconfiguration in the system software/services or malicious due to an intentional external attack. In the case of a malicious attack, the exact type of attack and the mitigation strategies to alleviate its harmful effects on the system operation.

V. System model

The system model in this work is a virtualized cloud data center running a collection of VMs leased by a set of k tenants. (refer to Figure 1).

The NodeGuard approach includes an algorithm to detect malfunctioning VM nodes by initiating a set of recursive probing mechanisms on the individual tenant VMs in the data center. The proposed algorithm leverages the VMI intrusion detection approach; however, instead of checking the operation of each VM individually, NodeGuard (i) detects the VMs that exhibit a processing error in computing a particular task or a time delay in executing this task (beyond the tenant-specific SLA constraints) in a logarithmic time complexity in terms of the size of the nodes, and (ii) the malicious or misbehaving VMs detected are deeply inspected by the VMI mechanisms for designating the exact source of the malfunction and whether this malfunction is a result of benign or malicious causes.

A. NodeGuard *algorithm*

1) Notation: The algorithm verifies the integrity and timeliness of the data processing. The domain of this algorithm is the VMs. The following notations and functions will be used in the algorithm description:

- $T = \{t_1, \ldots, t_k\}$ is the set of k tenants renting VMs in the data center ordered in decreasing order of number of VMs.
- T_{max} is the tenant with the maximum number of VM nodes
- $VM_i = \{v_1^i, \ldots, v_r^i\}$ is the set of r VMs belonging to tenant i.
- n is the total number of VMs in the data center.
- $Class = \{c_1, \ldots, c_p\}$ is the set of classes of distributed algorithms that can be used for probing the integrity of VM processing. An example distributed algorithm class that we will use, in the implementation part, is the matrix multiplication distributed algorithm based on the standard divide-and-conquer technique. It is worth mentioning here that the number of the distributed algorithm classes is an implementation-dependent configuration property that has a major impact on the security of the intrusion detection system. As the number of distributed algorithm classes increases the probability of the monitored VMs detecting the probing mechanisms decreases. The confusion introduced by selecting a different probing task in each protocol run enhances the transparency of the probing algorithm to be perceived as a normal processing task in the data center.
- $D = \{1, \ldots, d\}$ is a strictly increasing ordered set of the available size of subtasks comprising the distributed algorithms.
- $\Psi_{x,s}$ is a 2-dimensional vector referring to the distributed algorithm corresponding to the class $x \in Class$ with degree $s \in D$ in the pool of all available distributed probing algorithms.
- $\Psi_{x,s}(1, \ldots, s)$: is the set of s subtasks comprising $\Psi_{x,s}$. $\Psi_{x,s}$ contains a reference $(I(\Psi_{x,s}), H(F(\Psi_{x,s})), \tau_{\min} \leq \tau \leq \tau_{\max})$ tuple complying with the subtask implementation. $I(\Psi_{x,s})$ is the input parameters to the distributed task $\Psi_{x,s}$, $H(\cdot)$ is a one-way collision-resistant hash function, $F(\Psi_{x,s})$ is the output resulting from the execution of $\Psi_{x,s}$, and $\tau_{\min} \leq \tau \leq \tau_{\max}$ is the acceptable range of execution times needed to complete the processing of $\Psi_{x,s}$. More details about the one-way collision-resistant hash function can be found in[20].
- MD is an array of length $k-1$ for storing the hashes (digests) of the output. TS is an array of length $k-1$ for storing the digest timestamps. t_{start} is the timestamp indicating the start of task execution on the probed VMs.
- $Checked(i)$ is a Boolean function that returns true if the tenant $i \in T$ is checked for malicious behavior in a probing period.
- N is the parameter that represents the base case size to stop the recursion. It indicates the number of VMs that the algorithm converges to raise a misbehaving output signal. By default $N = 1$, but the value could be increased to enhance the convergence time to the algorithm at the expense of a less precise misbehaving VM localization.
- $result$ is the variable that represents the result of the distributed task by aggregating the output of the subtasks from the various tenant VMs.
- $VMI\ Interposition$ is the VMI primitive that allows the VMI to execute specific tasks on a particular VM in the data center.
- $Conquer$ is the function that accumulates the processing subtask output of the distributed task from the tenant VMs and aggregates it in a result variable.
- $Output$ is the function that computes the result of a subtask execution on a particular tenant VM.
- $Load$ is the function that loads a variable from the VM memory.
- $store$ is the function that stores a variable in the VM memory

2) Algorithm description: The complete algorithm specification is presented in Algorithm 1. The algorithm represents a function that takes three parameters as an input: The tenant to

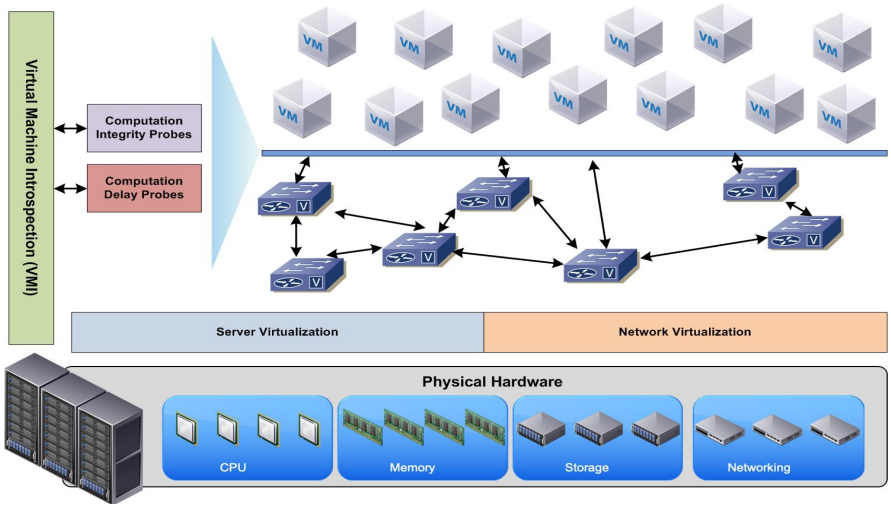

Figure 1: System model.

be currently inspected ($Ctenant$), the VM set of the current tenant ($VM_{Ctenant}(a,\ldots,r)$), and the type of misbehaving inspection required which can be either "processing" or "delay" ($type$). Accordingly, as an output, it results in the VM nodes misbehaving in processing or timely delivery of results.

The main qualities verified by the algorithm are (i) the accuracy and validity of the processing operations executed by the VMs and (ii) the rationality of the processing delay incurred when the tenant VMs execute a particular distributed task. In other words, the algorithm detects and locates any VM producing invalid computation results or incurring irrational or unacceptable processing delays. The specification of the quality to be tested by the algorithm is indicated using the type argument that can take a "processing" or "delay" value.

The probing system starts by randomly selecting a certain type (Class) of distributed algorithms that comprises several subtasks (Degree) that maps to the number of VM's in the probed group. As discussed earlier, the random selection of the distributed class aids in picking a different class in each protocol run, thus, decreasing the probability of the monitored VMs detecting the probing nature of the task. The algorithm follows a recursive divide-and-conquer design that inspects the VMs of each tenant separately starting with the tenant leasing the largest number of VMs.

Selecting the tenant to start the probing process with is a design choice that needs to be determined. The algorithm can possibly select a tenant at random to start inspecting or it can select it based on a predefined criterion. Without loss of generality, we chose to start with the tenant with the largest number of VM's. We faithfully believe that this decision can statistically ensure the timely detection of misbehaving VMs since such tenants are probabilistically more prone to including misconfigured VMs due to their relatively large VM base.

The algorithm to inspect the VM set of the tenants is achieved using a simple ***for loop*** that iterates over the various tenant networks and applies the NodeGuard function on each tenant VM set respectively, as NODEGUARD ($T[i], VM_{Ctenant}(1,\ldots,r)$, "$processing$"), $\forall i = 1,\ldots,k$ and such that $Checked(T[i])$ is false.

As stated in Algorithm 1, the NODEGUARD is a recursive function. It accepts as inputs: (1) the tenant to be currently inspected, (2) the VM set of the current tenant ($Ctenant$), and the type of misbehaving inspection required which can be either "processing" or "delay".

The base case of the NODEGUARD function is reached when the probed network consists of *N* remaining VM nodes ($N = 1$ by default) and the algorithm converges to locate the individual VM node (or set of nodes) instigating the misbehaving processing or delay of the assigned tasks. (refer to lines 3–5).

In lines 6–10, we select the tenant with the maximum number of VMs.

Line 11 saves the start time of the probing mechanism which will aid in calculating the delay in the later phases of the algorithm.

The distributed tasks to be executed on the VMs of the current tenant under inspection are selected randomly from the pool of distributed tasks and assigned to the VMs in $VM_{Ctenant}$. Each VM in the set $VM_{Ctenant}$ computes the assigned task through VMI interposition. Note that, the operation " $(task, input) \longleftrightarrow VM$ " indicates a task with its corresponding input set is to be executed on the VM using VMI interposition. (refer to lines 12–15).

The subtask processing results are sent to all other tenants in the data center. More precisely to one randomly selected VM in each tenant group other than the currently inspected tenant ($Ctenant$). (refer to lines 16–17).

Each node in the set $VM_{Ctenant}$: (i) receives the subtask processing result from each VM in the set $VM_{Ctenant}$ (ii) accumulates the final result using the particularities of the probing distributed algorithm, and (iii) hashes the final result to produce the resulting digest. (refer to lines 18–19).

The digest from each tenant along with the digest generation timestamp are respectively stored in the ***MD*** and ***TS*** vectors at index t. (refer to lines 20–25).

The resulting $(k-1)$-element MD is inspected with respect to the expected digest value $H(F(\Psi_{x,s}))$. if $H(F(\Psi_{x,s}))$ does not match any entry in the ***MD*** vector, this corroborates, with a high confidence level, the fact that the probed tenant $Ctenant$ comprises malicious VM nodes contributing to an invalid processing results in the final $\Psi_{x,s}$ output. In the latter case, the algorithm recursively splits the $Ctenant$ VM network into two equal logical domains and probes each half separately using the same logic as described above but with different distributed algorithm class and degree and using the "processing" type argument to the recursive function. (refer to lines 26–28).

On the other hand, if there is a match of at least one entry in MD with $H(F(\Psi_{x,s}))$, then this indicates that the current tenant VMs are correctly behaving. In this latter case, tenants corresponding to the entries with the nonmatching values in the ***MD*** vector are assigned next in row for probing using the "processing" type argument to the recursive function. (refer to lines 30–31).

Analogously, the algorithm checks any tenant misbehavior represented in excessive processing delay by inspecting the ***TS*** algorithm and subtracting the start time of the $\Psi_{x,s}$ execution from all the ***TS*** $k-1$ entries. Note that only the tenant ***TS*** indexes mapping to correct digest result calculation in the ***MD*** vector are probed for the time delay. If the time delays represented by all the TS vector entries checked are outside the range of acceptable times τ, then this indicates, with a high confidence level, that $Ctenant$ is causing this delay. Accordingly, the algorithm recursively splits $VM_{Ctenant}$ into two equal logical domains and applies the probing operations on each part using the "delay" type argument. (refer to lines 32–35).

If at least one timestamp entry in ***TS*** is producing time delays within the range τ, then this indicates that the probed $Ctenant$ is most probably complying with the acceptable time delay range and thus all non-checked tenants mapping to out of range delay entries in ***TS*** are probed using the "delay" type argument. (refer to lines 37–39).

Any VM domain probed producing an invalid processing result or an excessive processing delay undergoes the same division process until the recursive base case is reached.

B. Algorithmic runtime complexity

The dominant input size contributing to the main complexity of the algorithm is represented in n, the total number of VMs in the data center. We assume that n is significantly higher than the number of tenants k. As discussed previously, the NODEGUARD algorithm recursively operates on the VM set of each tenant in a divide-and-conquer fashion to locate the VM node or set of nodes exhibiting a misbehaving execution or instigating unreasonable delay in providing the requested services. Let $T(n)$ be the total runtime execution function of the NODEGUARD algorithm. Then,

$$T(n) = \left\{ \sum_{i=1}^{k} \mathcal{O}(f(|VM_k|)), n = \sum_{i=1}^{k} |VM_k| \right\} \quad (1)$$

where $\mathcal{O}$ is the asymptotic worst-case runtime classification function, $|VM_k|$ is the number of VMs belonging to the k-th cloud tenant, and f is the runtime complexity function on the k-th cloud tenant.

When the number of misbehaving VM nodes at a particular tenant k is low, this number is considered of order $\mathcal{O}(n)$, i.e. a constant in algorithmic complexity terms, compared to the total number of VMs $|VM_k|$, and as a result in the order of n, $f(|VM_k|)$ asymptotically approaches $\log_2(|VM_k|)$. This can be easily proved by representing $f(|VM_k|)$ in its recursive form and solving the resulting difference relation to find its closed-form as a function of $|VM_k|$. The difference relations for $f(|VM_k|)$ is presented as follows:

$$f(|VM_k|) = \begin{cases} f(\frac{|VM_k|}{2}), & |VM_k| > 1 \\ 1, & |VM_k| = 1 \end{cases}$$
$$\Rightarrow f(|VM_k|) \in \mathcal{O}(\log_2(|VM_k|)) \quad (2)$$

On the other hand, when the misbehaving rate at tenant k is relatively high, say in the order of $|VM_k|$, and as a result, n, which in the worst-case scenario indicates that the number of misbehaving VM nodes asymptotically approaches the total number of VMs $|VM_k|$ in the tenant k jurisdiction, then the complexity of the NODEGUARD algorithm asymptotically approaches $\mathcal{O}(|VM_k|)$. In such a case, the two halves of the tenant VM network signal a malicious operation, which designates the scenario when the recursive divide-and-conquer technique is required to probe both halves in all the recursive steps of the algorithm. This results in the following recurrence relation for $f(|VM_k|)$:

$$f(|VM_k|) = \begin{cases} 2\, f(\frac{|VM_k|}{2}) + \mathcal{O}(1), & |VM_k| > 1 \\ 1, & |VM_k| = 1 \end{cases}$$
$$\Rightarrow f(|VM_k|) \in \mathcal{O}(|VM_k|) \quad (3)$$

This scenario, of having all the VMs in the set $|VM_k|$ malicious, is assumed uncommon in conventional cloud environments, and the $\mathcal{O}(|VM_k|)$ runtime complexity is exactly the runtime of the traditional VMI algorithm that inspects each VM individually to detect malicious VM behavior. The $\mathcal{O}(\log_2(|VM_k|))$ runtime complexity results in a total runtime $T(n)$ as follows:

$$T(n) = \left\{ \sum_{i=1}^{k} \mathcal{O}(\log_2(|VM_k|)), n = \sum_{i=1}^{k} (|VM_k|) \right\}$$
$$= \left\{ \mathcal{O}(\log_2(\{\max_{i=1,\ldots,k}(|VM_i|)\}), n = \sum_{i=1}^{k} (|VM_i|) \right\}$$
$$\Rightarrow T(n) \in \mathcal{O}(\log_2(n)) \quad (4)$$

A similar analysis for the distrustful scenario of having all VM nodes in the tenant networks exhibiting a misbehaving

Algorithm 1: NODEGUARD algorithm

Function NODEGUARD ($Ctenant$, $VM_{Ctenant}(a,\ldots,r)$, *type*) **:**
- Set $Ctenant$ as $Checked$
- $N = 1$
- **if** $a - r + 1 = N$ **then**
  - **return** $VM[a], VM[a+1], \ldots, VM[a+r]$ as a "type" maliciously detected node(s)
- Select random $\Psi_{x,s}$, s.t.: $x =$ random$(1,\ldots,p)$ AND $s = 1$
- **for** $i = 1$ *to* d **do**
  - **if** $Degree[i] \geq Tmax$ **then**
    - $s = i$
    - **break**
- set t_{start}
- **for** $i \leftarrow 1$ *to* s **do**
  - $j \leftarrow (i-1) \mod (|VM_{Ctenant}|) + 1$
  - Interposition @ $VM_{Ctenant}[j]$
  - $(\Psi_{x,s}[i], (I(\Psi_{x,s})) \longleftrightarrow VM_{Ctenant}[j]$
  - **for** t *in* $T(1,\ldots,k)$ *AND* $t \neq Ctenant$ **do**
    - send $Output(\Psi_{x,s})[i]$, $VM_{Ctenant}[j])$ to any running VM in VM_t
    - Interposition @ VM_t
    - $Conquer(result, Output(\Psi_{x,s}[i]), VM_{Ctenant}[j])$
- **for** t *in* $T(1,\ldots,k)$ *AND* $t \neq Ctenant$ **do**
  - Interposition @ VM_t
  - $digest = H(result)$
  - $Load(digest)$
  - $Store(MD[t], digest)$
  - $Store(TS[t], digest_{timestamp})$
- **if** $H(F(\Psi_{x,s})) \notin MD$ **then**
  - NODEGUARD $(Ctenant, VM_{Ctenant}(a,\ldots,r/2)$, "$processing$")
  - NODEGUARD $(Ctenant, VM_{Ctenant}((r/2)+1,\ldots,r)$, "$processing$")
- **else**
  - **for** *all* t *where* $MD[t] \neq H(F(\Psi_{x,s})$ *AND* $!Checked(T[t])$ **do**
    - NODEGUARD $(t, VM_t(1,\ldots,r)$, "$processing$")
- **for** *all* t *where* $MD[t] = H(F(\Psi_{x,s})$ **do**
  - **if** *all* $TS[t] - t_{start} \notin \tau$ **then**
    - NODEGUARD $(Ctenant, VM_{Ctenant}(a,\ldots,r/2)$, "$delay$")
    - NODEGUARD $(Ctenant, VM_{Ctenant}((r/2)+1,\ldots,r)$, "$delay$")
  - **else**
    - **for** *all* t *where* $MD[t] = H(F(\Psi_{x,s})$ *AND* $!Checked(T[t])$ **do**
      - **if** $TS[t] - t_{start} \notin \tau$ **then**
        - NODEGUARD $(t, VM_t(1,\ldots,r)$, "$delay$")

processing or delay patterns results in an asymptotic runtime complexity of $T(n) \in \mathcal{O}(n)$ which is the same runtime complexity of the traditional VMI intrusion detection algorithm inspecting the VMs in the data center one at a time.

VI. PERFORMANCE EVALUATION

A proof-of-concept testbed emulation of the system design is implemented on the Mininet network emulator [21]. Mininet enables the creation of a real virtual network composed of a central controller and a set of hosts and switches running actual Linux kernel on a single computer. Mininet emulates an actual cloud environment with virtualized software-defined networking (SDN) modules that allow for the realization and testing of relatively large-scale cloud services. Selecting Mininet for implementing the NODEGUARD system model is not only due to the realistic cloud platform and scalable virtualization support but also due to its seamless ability to simulate the introspection primitives provided by a standard VMI platform. This is done by leveraging the centralized control modules realized in the SDN controller to have access to the underlying processing units in the virtualized hosts. This is because, in Mininet, The SDN controller is deployed on the local host system running the network virtualization hypervisor with direct access to the VM runtimes and memory resources. We employed a VMware Linux VM for running the Mininet network emulator. The VM is hosted on VMware Fusion Professional Version 12.1.0 running on a MacBook Pro, Intel Core i9 @ 2.4GHz, with 64GB of main memory. The guest system was assigned 4 cores, 33.5GB of memory,

and it was running on Ubuntu 14.04 (64 bit), using Floodlight v1.2.

The system configuration we followed is comprised of 8 cloud tenants (T1 to T8) leasing several VMs, ***r***, as indicated by Table I. This implies a network size n of 3875 VMs. This is the largest number of virtual hosts that we were able to boot on a single emulation laptop with the above-listed configuration. To emulate the malicious behavior in the network, we induced intentional processing flaws in a set of VMs at each tenant configuration. We formalized the degree of malicious behavior in the network by the parameter ρ which indicates the percent maliciousness in the network. ρ is defined as the percent of VM nodes exhibiting misbehaving or malicious behavior to the total number of VM nodes at a particular tenant configuration. The misbehaving pattern could be designated by an invalid processing result when executing a corresponding probing task or an intolerable processing delay in generating the result. In the testbed implementation, we selected 12 ρ malicious rates starting with 0% maliciousness indicating an error-free network to 100% where uncommonly all the VM nodes in the network are demonstrating a form of malicious behavior. The NODEGUARD algorithm convergence time for each network malicious rate is computing and compared to that of the traditional VMI intrusion detection methodology. Without loss of generality, we utilized a set of matrix multiplication distributed tasks to probe the different tenant VMs.

In the traditional VMI technique, the input matrix sizes are set to 100x100 elements of random values in the range of [1, 500000]. In the NODEGUARD approach, to ensure a fair comparison with the traditional approach, we selected the parent problem size in a way to ensure that each subtask consists of the multiplication of input matrices of 100x100 elements. We applied the standard divide-and-conquer matrix multiplication algorithm to subdivide the large matrix multiplication problem into a set of smaller subtasks to respectively execute at each tenant VM as designated by the NODEGUARD algorithm specifications. The NODEGUARD probing mechanisms are periodically replicated every 10 minutes over a period of 7 days. We selected a random malicious rate ρ discretely from the set [0, 5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]. The average runtime for completing a NODEGUARD probing step is calculated for the 12 different malicious rates ρ and this is respectively compared to the runtime of executing the standard VMI intrusion detection algorithm that checks each VM individually for correct processing and reasonable delay. The results in Figure 2 remarkably comply with the theoretical analysis presented in the previous section. As expected, when the malicious rate is relatively low, the NODEGUARD algorithm exhibits a runtime complexity in the order of $\log_2(n)$. As the malicious rate increases above the 50% - 60% mark, the runtime complexity incrementally approaches a linear function and gets closer to the performance of the traditional VMI intrusion detection technique. Due to the time limitation for presenting this work, the probing runtime measurements are carried out on idle VMs without any accompanying processing workload. Future extensions will consider the effect of various

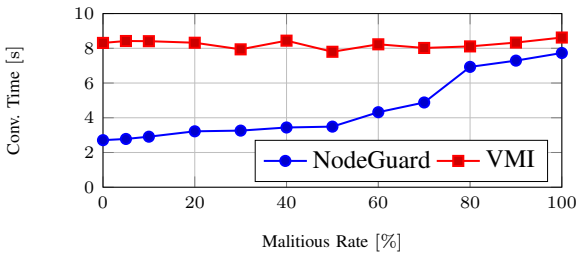

Figure 2: Convergence time of the NODEGUARD approach compared to that of the standard VMI intrusion detection method for several VM malicious rates.

probabilistic workloads on the performance of the probing mechanisms on the tenants' VMs. The range of reasonable processing times $\tau_{\min} \leq \tau \leq \tau_{\max}$ is incrementally determined from one probing period to the next based on the execution time of the distributed task on the tenant with the maximum number of VMs (this determines $\tau_{\max}$) and the tenant with the minimum number of VMs (this determines $\tau_{\min}$) in each probing period. Since the values of $\tau_{\min}$ and $\tau_{\max}$ depend on the overall processing time of each distributed task $TS[t] - t_{\text{start}}$ on a particular VM tenant network t (which can change from one probing period to the other (even within an individual probing period), the values of $t_{\min}$ and $t_{\max}$ dynamically change based on this variation in the execution time. To sustain a smooth disparity in the values of $t_{\min}$ and $t_{\max}$ we followed an algorithm analogous to the retransmission timer derivation algorithm maintained in TCP [22].

The details of the $t_{\min}$ and $t_{\max}$ calculations are provided in the subsequent smoothing equations using the estimators $m\tau$ and $v\tau$, respectively, representing the mean and variance of the execution time of the distributed task in the k^{th} probing period.

We start with $\tau_{\max}$, let $t_{\max}$ represent the tenant with the maximum number of leased VMs and $t_{\min}$ represent the tenant with the minimum number of leased VMs:

$$m\tau_{k+1} = \alpha\,(TS[t_{\max}] - t_{\text{start}}) + (1-\alpha)\,m\tau_k$$
$$v\tau_{k+1} = \beta\,(|TS[t_{\max}] - t_{\text{start}} - m\tau_k|) + (1-\beta)\,v\tau_{k+1}$$
$$\tau_{\max} = m\tau_k + 4\,v\tau_{k+1}$$

In the first probing period, the mean and variance estimators are set as follows:

$$m\tau_1 = TS[t_{\max}] - t_{\text{start}}, \quad v\tau_1 = \frac{TS[t_{\max}] - t_{\text{start}}}{2}$$

Analogously, $\tau_{\min}$ is derived as follows:

$$m\tau_{k+1} = \alpha\,(TS[t_{\min}] - t_{\text{start}}) + (1-\alpha)\,m\tau_k$$
$$v\tau_{k+1} = \beta\,(|TS[t_{\min}] - t_{\text{start}} - m\tau_k|) + (1-\beta)\,v\tau_{k+1}$$
$$\tau_{\min} = m\tau_k + 4\,v\tau_{k+1}$$

In the first probing period, the mean and variance estimators are set as follows:

$$m\tau_1 = TS[t_{\min}] - t_{\text{start}}, \quad v\tau_1 = \frac{TS[t_{\min}] - t_{\text{start}}}{2}$$

Table I: Samples of VMs configurations.

Tenant	T1	T2	T3	T4	T5	T6	T7	T8
Number of VMs *r*	1250	1000	750	500	200	100	50	25

The gains α and β are set to $\frac{1}{4}$ and $\frac{1}{8}$ respectively [22].

VII. Conclusion and future directives

In this work, we presented NodeGuard, a security approach for detecting and isolating misbehaving processing nodes in virtualized cloud data centers. The main contribution of this work is represented by employing the VMI interposition and introspection functions to detect and isolate any source of maliciousness in the tenants' VMs. NodeGuard follows a divide-and-conquer strategy to result in a logarithmic run time complexity compared to the linear traditional VMI intrusion detection mechanisms. The system is implemented in a virtualized cloud environment using the Mininet network emulator. Experimental performance measurements corroborate the logarithmic complexity advantage of NodeGuard over the traditional VMI intrusion detection strategy. Future extensions include: (1) investigating customized divide-and-conquer configurations that might take advantage of policy-based network division mechanisms to better enhance the runtime complexity of the system, and (2) realizing the system in a real cloud computing environment to get a more realistic performance profiling of the security processing.

References

[1] A. Celesti, M. Fazio, A. Galletta, L. Carnevale, J. Wan, and M. Villari, "An approach for the secure management of hybrid cloud–edge environments," *Future Generation Computer Systems*, vol. 90, pp. 1–19, 2019.

[2] B. D. Payne, "Simplifying virtual machine introspection using libvmi," *Sandia report*, pp. 43–44, 2012.

[3] R. Wu, P. Chen, P. Liu, and B. Mao, "System call redirection: A practical approach to meeting real-world virtual machine introspection needs," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2014, pp. 574–585.

[4] B. Taubmann, N. Rakotondravony, and H. P. Reiser, "Cloudphylactor: Harnessing mandatory access control for virtual machine introspection in cloud data centers," in *2016 IEEE Trustcom/BigDataSE/ISPA*. IEEE, 2016, pp. 957–964.

[5] A. Al-Dulaimy, R. Zantout, W. Itani, and A. Zekri, "Job submission in the cloud: energy aware approaches," in *Proceedings of the World Congress on Engineering and Computer Science*, vol. 1, 2016.

[6] T. Garfinkel, M. Rosenblum *et al.*, "A virtual machine introspection based architecture for intrusion detection." in *Ndss*, vol. 3, no. 2003. Citeseer, 2003, pp. 191–206.

[10] B. Taubmann and B. Kolosnjaji, "Architecture for resource-aware vmi-based cloud malware analysis," in *Proceedings of the 4th Workshop on Security in Highly Connected IT Systems*, 2017, pp. 43–48.

[7] B. Taubmann and H. P. Reiser, "Towards hypervisor support for enhancing the performance of virtual machine introspection," in *IFIP International Conference on Distributed Applications and Interoperable Systems*. Springer, 2020, pp. 41–54.

[8] F. Westphal, S. Axelsson, C. Neuhaus, and A. Polze, "Vmi-pl: A monitoring language for virtual platforms using virtual machine introspection," *Digital Investigation*, vol. 11, pp. S85–S94, 2014.

[9] P. Mishra, V. Varadharajan, E. S. Pilli, and U. Tupakula, "Vmguard: A vmi-based security architecture for intrusion detection in cloud environment," *IEEE Transactions on Cloud Computing*, vol. 8, no. 3, pp. 957–971, 2018.

[11] V. Varadharajan and U. Tupakula, "On the design and implementation of an integrated security architecture for cloud with improved resilience," *IEEE Transactions on Cloud Computing*, vol. 5, no. 3, pp. 375–389, 2016.

[12] L. Jia, M. Zhu, and B. Tu, "T-vmi: Trusted virtual machine introspection in cloud environments," in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2017, pp. 478–487.

[13] F. Fargo, O. Franza, C. Tunc, and S. Hariri, "Vm introspection-based allowlisting for iaas," in *2020 7th International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*. IEEE, 2020, pp. 1–4.

[14] B. Borisaniya and D. Patel, "Towards virtual machine introspection based security framework for cloud," *Sādhanā*, vol. 44, no. 2, p. 34, 2019.

[15] A. Melvin, G. J. Kathrine, and J. I. Johnraja, "The practicality of using virtual machine introspection technique with machine learning algorithms for the detection of intrusions in cloud," 2021.

[16] T. K. Lengyel, S. Maresca, B. D. Payne, G. D. Webster, S. Vogl, and A. Kiayias, "Scalability, fidelity and stealth in the drakvuf dynamic malware analysis system," in *Proceedings of the 30th annual computer security applications conference*, 2014, pp. 386–395.

[17] P. Mishra, P. Aggarwal, A. Vidyarthi, P. Singh, B. Khan, H. H. Alhelou, and P. Siano, "Vmshield: Memory introspection-based malware detection to secure cloud-based services against stealthy attacks," *IEEE Transactions on Industrial Informatics*, vol. 17, no. 10, pp. 6754–6764, 2021.

[18] M. Shamseddine, W. Itani, A. Kayssi, and A. Chehab, "Virtualized network views for localizing misbehaving sources in sdn data planes," in *2017 IEEE International Conference on Communications (ICC)*. IEEE, 2017, pp. 1–7.

[19] M. Shamseddine, W. Itani, A. Chehab, and A. Kayssi, "Network programming and probabilistic sketching for securing the data plane," *Security and Communication Networks*, vol. 2018, 2018.

[20] S. Gueron, S. Johnson, and J. Walker, "Sha-512/256," in *2011 Eighth International Conference on Information Technology: New Generations*. IEEE, 2011, pp. 354–358.

[21] "Mininet," http://mininet.org/, May 2021.

[22] V. Paxson, M. Allman, J. Chu, and M. Sargent, "Computing tcp's retransmission timer," rfc 2988, November, Tech. Rep., 2000.