DCGUARD: A Holistic Approach for Detecting and Isolating Malicious Nodes in Cloud Data Centers

Wassim Itani, Maha Shamseddine, Auday Al-Dulaimy, Thomas Nolte, Alessandro V. Papadopoulos

***Abstract*—This paper presents DCGUARD, a unified security approach for detecting and isolating misbehaving computing and forwarding nodes in multi-tenant virtualized cloud data centers. DCGUARD employs technological advancements in Virtual Machine Introspection (VMI), Software-Defined Networking (SDN), and secure probabilistic sketching to detect and isolate parts of the Virtual Machines (VMs) and network switches experiencing malicious behavior dynamically. The main contribution lies in designing a divide-and-conquer strategy that utilizes VMI and network programmability to apply focused distributed task and packet probing mechanisms on portions of the data center network rather than focusing the security functions on the entire physical network. The processing VMs and network switches are recursively partitioned into independent logical groups inspected individually to localize abnormal/malicious computing and switching nodes incrementally. This remarkably enhances the efficiency of the detection mechanisms, which opportunistically approaches a logarithmic time complexity in the number of protocol steps towards convergence (compared to a linear time complexity in traditional intrusion detection systems) when a relatively low number of hostile VMs and switches are present. Real experiments are evaluated, and a test-bed blueprint of the proposed design is emulated in a virtualized cloud environment using the Mininet emulator. The performance, convergence, and accuracy benchmarks corroborate the analytical advantage of the proposed security approach.**

***Index Terms*—Cloud computing, Data center, Security, Virtual machine introspection, VMI, Software-Defined Networking, SDN.**

I. INTRODUCTION

Cloud computing architectures have expanded rapidly in the past decade due to its flexible pay-as-you-go service model and the "elastic" features of the advertised storage and processing services. Today, cloud computing offers software, infrastructure, and platform services, among others, that allow customers to scale their computing and storage resources up and down on demand to achieve cost-effective, minimal administrative service development, deployment, and operation [1]. The cloud computing advantage is mainly attributed to the virtualization technologies employed in the modern cloud computing data center. In this model, the VM is the main interface with customer services and the key computing unit responsible for service management and execution. The high reliance on VM nodes in the cloud computing model makes those nodes a natural attack target by malicious users and scripts, including the various malware types that may impose various modifications and delay attacks that surpass the Service Level Agreement (SLA) tolerance. In addition, VMs administratively belong to different tenant networks with diverse trust levels and security enforcement mechanisms. On the same front, the great advancements in network virtualization technologies enabled the creation of softwarized tenant networks consisting of virtual switching and forwarding nodes on top of the actual physical network. As in the case of VM computing nodes, network forwarding nodes carry out critical routing and switching functionalities in the cloud network and can also belong to different tenants administratively. Add to this the fact that virtualized forwarding nodes should be capable of handling large amounts of network traffic, some of which might be maliciously targeting specific strategic points in the network to cause disruption, denial of service, or man-in-the-middle attacks, to name a few.

This necessitates the development of a comprehensive security model to safeguard against VM vulnerabilities and malicious activities in network switching and forwarding. Moreover, the desired security model should be capable of handling benign misconfiguration vulnerabilities whose adverse effects are often not less catastrophic than the ramifications of intentional malicious attacks. The intended security model should scale to large multi-tenant cloud data centers, target diverse and extensive attack vectors, and operate efficiently to detect and isolate sources of maliciousness/misconfiguration in the computing and networking infrastructures. The great advancements in virtualization monitoring technologies, such as Virtual Machine Introspection (VMI), Software-Defined Networking (SDN), and network virtualization, have paved the way for an unprecedented set of tools that can be utilized to detect malicious activity in computing and networking operations. VMI is a technique for monitoring the runtime state of VMs and can be employed for several security objectives, such as malware and intrusion detection. SDN and network virtualization and some intelligent probabilistic sketching techniques can support the design of intelligent probing and intrusion detection techniques that can efficiently operate in sizable cloud data center networks and mitigate diverse and large-scale attacks. In this paper, we present DCGUARD, a unified security model that leverages the latest advancements in VMI, SDN, Network programmability, and secure probabilistic sketching to detect and isolate sources of maliciousness in the processing and networking nodes in multi-tenant virtualized cloud data centers.

The DCGUARD approach includes two main algorithms that work in parallel but independently to improve the system's

This work is supported by the Swedish Knowledge Foundation (KKS), and by the Swedish Research Council (VR).

W. Itani and M. Shamseddine are with George Mason University, USA.

A. Al-Dulaimy is with Mälardalen University and Dalarna University, Sweden.

T. Nolte and A.V. Papadopoulos are with Mälardalen University, Sweden.

performance: The first algorithm, called $_P$DCGUARD, detects malfunctioning virtual processing nodes by initiating a set of recursive probing mechanisms. It detects processing flaws in the computation of distributed tasks assigned to tenants' VMs. Processing flaws can be induced by malicious user processes running on the VMs, including the various types of malware that may impose a modification attack on the result of computation. Any modification on the input-processing-output pipeline would result in a processing flaw detected by the $_P$DCGUARD algorithm and is considered a malicious attempt to disrupt the computation of the distributed task. The second algorithm, called $_F$DCGUARD, reveals sources of maliciousness in the network. It identifies malicious forwarding nodes in the SDN data plane by applying a graph partitioning algorithm to create two halves of the network with minimal connecting edges and using OpenFlow controller messages to isolate the two partitions. The SDN controller forks a probe transmitter and receptor daemon processes with each half to respectively send and receive the probing packets to detect any suspicious activity in the forwarding nodes.

It is worth mentioning that DCGUARD does not assume the defense against a specific malware category or signature. Instead, it follows a generalized intrusion detection approach that relies on behavioral rather than signature-based techniques to detect malicious/misbehaving activities in the VM processing or network forwarding operations. Moreover, DCGUARD does not assume the ability to detect stealthy malware activities that do not stipulate the intrusion detection response specified in the DCGUARD algorithms presented later in this paper. For instance, the DCGUARD algorithms do not claim the ability to safeguard the VM network against malware that does not induce processing corruption or delays or modify the network traffic patterns and timely forwarding operations. A valuable resource on this topic can be found in [2].

In many contexts in this paper, we use the terms "malicious" and "misbehaving" to respectively denote the capability of the DCGUARD algorithms to detect intentional attacks on the network processing and forwarding resources as well as benign sub-optimal misconfigurations leading to vulnerabilities.

A. Contribution

The main contributions of this work are the following:

- Designing a divide-and-conquer strategy that utilizes VMI and network programmability to apply focused distributed task and packet probing mechanisms on portions of the data center network rather than focusing the security functions on the entire physical network.
- Recursively partitioning the set of processing VMs and network switches into independent logical groups inspected individually to localize abnormal/malicious computing and switching nodes incrementally. This remarkably enhances the efficiency of the detection mechanisms, which opportunistically approaches a logarithmic complexity in the number of protocol steps towards convergence (compared to a linear time complexity in traditional intrusion detection systems) when a relatively low number of hostile VMs and switches are present. The computational complexity and network traffic size are comparable to the traditional approach (linear time complexity) but with a major security advantage and better compliance with the modern cloud service model.
- Presenting a comprehensive mathematical and empirical performance analysis of the proposed security approach in its various stages when operating on the VM computing nodes and the network switches.
- Testing real experiments and developing a test-bed blueprint of the proposed design in a virtualized cloud environment using Mininet network emulator. The performance and accuracy benchmarks executed corroborate the analytical advantage of the proposed security approach.

Our previous work [3] inspires the computation security approach we present in this work. In [3], we presented the basic idea of designing a security approach using the VMI APIs to isolate misbehaving computational VMs in virtualized data centers. The main contribution provided by DCGUARD over [3] is summarized in designing a dedicated network security protocol to detect sources of maliciousness in forwarding nodes using the programmability and network virtualization capabilities of SDN and the efficient packet summarization features of probabilistic sketching. The newly added network security protocol complements the computational security protocol to provide a unified system for detecting misbehaving/malicious nodes in modern cloud networks.

B. Organization of the paper

The rest of this paper is organized as follows: Section III overviews the related work. Section IV presents two threat models: one for the adversary that may corrupt the processing logic executing on VMs and the other for the attacker that can execute active attacks on the network traffic. The proposed solution system design and performance evaluation are presented in Section V and Section VI, respectively. Conclusions are presented in Section VII.

II. BACKGROUND

A. Virtual Machine Introspection (VMI)

Virtualization technology is the foundation of cloud computing [4]. It is enabled by a hypervisor, also known as Virtual Machine Manager (VMM), a software that virtualizes all hardware resources, allowing multiple VMs to multiplex the Physical Machine's (PM) resources transparently. VMM is the mechanism that facilitates the construction of VMI. As stated in [5], VMI can leverage three main properties of VMMs: ***Isolation:*** to ensure that any software running inside a VM cannot access software running inside another VM or in the VMM, ***Inspection:*** meaning that the VMMs can access a VM's state, including the 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 ***Interposition:*** meaning that the VMM can interpose on certain VM operations (*e.g.,* executing privileged instructions). Leveraging these properties is essential for the proposed algorithms. The original VMI architecture was presented in [5] 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 VM state, ***Monitor commands*** to know when certain events occur and request notification through an event delivery mechanism, and ***Administrative commands*** to control the VM execution.

In $_P$DCGUARD, VMI interposition APIs are employed to execute remote distributed probing tasks on tenant VMs to stand on the correct and timely processing capabilities of these VMs. The VMI inspection APIs are used to assess and verify the validity of the computation results and their timely availability by inspecting specialized hashing and timestamp data structures on the tenant VMs. It should be noted here that reference implementations for VMI modules are available, such as the popular LibVMI library [6], which provides a C-based engine with Python bindings to support the VMI mentioned above commands. LibVMI has a wide platform support. It is compatible with the Xen, KVM, and Qemu virtualization platforms and supports both Linux and Windows VMs. Regarding architectural support, LibVMI is designed to run on 32-bit, PAE, and 64-bit x86 and ARM Cortex-A15 architectures. LibVMI provides efficient, near-native speed, VMI command performance using low-level and high-level API calls. In other words, VMI command calls performed outside the VM induce roughly the same execution times compared to commands running inside the guest VM. Both inspection and interposition VMI calls run in the order of a few μsecs using low-level API calls and a few 10's of μsecs using high-level API calls (in exception to the initial VMI initialization operation, which runs in the order of a few $m$secs) [7]. This makes VMI modules highly portable and efficient for modern cloud data center service implementations.

B. Software-Defined Networking (SDN)

SDN is a network approach that uses software controllers to communicate with hardware infrastructure, aiming to direct network traffic efficiently. It makes networks programmable and more flexible regarding configuration, monitoring, and performance [8]. To detect malicious forwarding nodes in the network, we utilize network programming to partition the SDN data plane recursively. The SDN controller is responsible for specifying routing and forwarding rules network-wide through OpenFlow messages to the forwarding network switches. This network-wide control by the SDN controller allows for the creation of isolated network partitions, which would recursively map to a possible set of malicious switches in the network. To localize maliciously behaving switches, a graph-theoretic partitioning algorithm recursively divides the network into two equal-cardinality partitions with minimal interconnecting edges. A probing module on top of the SDN controller dynamically generates focused packets that probe each partition.

C. Sketching algorithms

Sketching algorithms, also called probabilistic sketching algorithms, are a set of mechanisms that allow the efficient processing of massive data sets using the compact space requirements of the underlying data structures. Sketching algorithms were the main design choice in this work due to their compact data summarization capabilities. These algorithms aid in efficiently detecting data value deviation in real time. This design choice of using probabilistic sketching data structures is related to the nature of the intrusion detection problem, which utilizes sizeable network data streams and timestamps to ensure the accurate detection of misbehaving switches along the recursively generated network partitions. Probabilistic sketching in such a framework results in a substantial reduction in computational run-time complexity, better storage utilization, and hence, a resource-efficient real-time detection of malicious forwarding nodes.

The probing mechanism employs two probabilistic sketching data structures: The Tug-of-War Sketch [9] and the Timestamp Accumulator [10].

III. Related work

A. Computation security approaches

As the VMI approach has high overhead [11], [6], many works are proposed to minimize such overhead. The work in [12] tried to reduce the performance overhead by integrating some VMI operations into the hypervisor. The authors in [13] leveraged the most commonly used VMI techniques to monitor the VM's status by executing VMI analysis scripts in the hypervisor domain. In [5], the authors presented the VMI approach that utilizes the host-based IDS and expands it outside the host to maximize 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 [14] employed VMI and Machine Learning techniques at the VMM to present a security architecture, called *VMGuard*, aiming to detect hidden malware by performing memory introspection. The work in [15] 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 [16], an integrated security architecture was presented. The architecture utilized some features from the VMs, such as the resources dedicated to VMs, types of 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. The work in [17] presented a VMI monitor approach, called T-VMI, to ensure the security of a specific VM in a certain host. The authors in [18] utilized 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 runtime phase to decide on the valid actions. In [19], a VMI-based security framework architecture was presented. It models the behavior

of processes running on VMs and uses multi-threading to analyze VMs' activities and control their events. The work in [20] proposed a model for detecting container malware. It exports memory snapshots via VMI in the containers running in VMs, reconstructs container semantics from these snapshots, and analyzes the binary execution information of container runtime.

The works mentioned above focus on securing the computation, not enhancing the time complexity while detecting intrusions. Generally, the main performance parameters that govern the design of efficient intrusion detection schemes are (1) the time needed by the protocol to detect and isolate the malicious/misbehaving nodes in the network (protocol convergence time) and (2) the number of protocol iteration/recursion steps needed to achieve the detection and isolation mechanisms. Those performance parameters mainly depend on the number of VMs in the cloud network, which is considered the dominant input variant influencing the time complexity of the intrusion detection protocols. Regarding the accuracy of the detection and isolation mechanism, two central metrics are generally analyzed: (1) the true positives probability and (2) the false positives probability. Receiver Operating Characteristics (ROC) curves provide an extrapolation of the balance between the successful attack detection of the intrusion detection algorithm and the false positive alarms generated.

Several works utilize the hypervisors to enhance the Performance of VMI. In [12], the authors explained how the performance could be improved by integrating some minor extensions for VMI operations into the hypervisor. The work in [21] incorporated VMI and Xen hypervisor for dynamic Windows malware analysis. However, such works require changes to the hypervisors, which is not a trivial task.

B. Networking security approaches

Recently, there has been a significant shift in security solutions, with a growing focus on SDN-based security for virtual networks. This shift is driven by the fact that SDN provides centralized control, dynamic network configuration, network traffic control, and network programmability, all of which contribute to creating more secure networks.

In [22], the authors introduced a novel security approach for SDN. This approach is designed to detect a range of attack situations, including network scanning attacks, OpenFlow flooding attacks, switch compromised attacks, and ARP attacks. The approach operates in both the data plane and control plane, and it leverages multiple observations of a Hidden Markov Model (HMM) to assess the security situation of the network. In [23], the authors presented an approach, called *CLOUDWATCHER*, that can direct network packets to be automatically inspected using pre-installed network security devices. The authors in [24] presented the *NetTSecVisor* approach that utilizes SDN for network management, aiming at enabling network security protection. The work in [25] proposed a scalable mechanism, called *NetFuse*, to protect against traffic overload in data center networks, with minimum measurement overhead. In [26], the authors proposed an anti-DoS approach based on SDN, called *DCPortalsNg*, to isolate virtual network traffic, direct network packets, and thus control which VM can be reached by specific packet(s). Some works used Neural Networks [27] and deep learning [28] for intrusion detection in SDN networks. In our previous works [29], [30], 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 are not designed to secure data processing in VMs. The work in [31] proposed an approach to synchronize VMs with a discrete-event network simulator. The approach ensures that packets transmitted by the VMs are received by their intended destination at the exact time calculated by the simulator, with network-based timings remaining unaffected by any possible VMI pause.

A promising research field worth mentioning here is the reliance on multi-domain SDN settings using collaborative controllers for securing networks supporting programmable components [32]. This research mainly focuses on managing large-scale DDoS attacks on programmable SDN-based networks [33]. A comprehensive survey on this research field is presented in [34]. Note that the $_F$DCGUARD network security protocols can contribute to the security of multi-domain programmable SDN networks and benefit from the collaborative controller model due to three main reasons: (1) $_F$DCGUARD is designed to operate in expansive multi-tenant cloud networks supporting programmable SDN components. Employing multi-domain controller settings would provide better resource management and efficiency in targeting the network slices that require more security inspection and analysis. (2) By design, the $_F$DCGUARD protocol engine is aimed at targeting large-scale security attacks, which is corroborated by the divide-and-conquer strategies employed and the reliance on efficient probabilistic sketching schemes to manage high network traffic rates. Hence, multi-domain SDN settings can play a major role in enhancing the detection efficiency of large-scale attack scenarios and decreasing the protocol convergence time. (3) The main focus in today's security research in multi-domain collaborative SDN architectures is handling large-scale DoS attacks. Extending the $_F$DCGUARD to operate in multi-domain SDN settings using collaborative controllers can contribute to the detection of a wide array of attack vectors as specified in the network forwarding attacker model presented in Section IV-B. We leave porting the presented network security protocols to multi-domain SDN architectures for a future extension.

IV. THREAT MODEL

The threat model we assume in this work targets two main functions in the cloud data center: (1) the VM processing attacker model and (2) the network forwarding attacker model. Both attacker models assume a generalized intrusion detection approach that relies on behavioral rather than signature-based techniques to detect misbehavior in the network.

A. The VM Processing Attacker Model

The threat model is represented by an attacker model that can 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, shared libraries, and APIs running on the VM or the guest VM OS. In such a scenario, the attack could leverage the malicious services of a virus, trojan horse, or any malware that can disrupt the normal operation of the processing software, the supporting libraries, interfaces, or the underlying OS. Moreover, the attacker can execute 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 and the interpretation of the probing output reside in the VMI module, which is assumed safe and isolated from the cloud's 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 model is designed to periodically initiate a set of well-specified processing probing tasks on the tenants' VMs and leverage VMI introspection and interposition to ensure 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 indistinguishable probing mechanisms that are viewed as any normal processing tasks to the tenant VMs. This is why, as will be shown later in Section V, 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 VMI system deeply inspects the respective nodes to determine whether it is benign due to a misconfiguration in the system software/services or malicious due to an intentional external attack.

B. The Network Forwarding Attacker Model

The network forwarding model we employ is based on the SDN architecture. This model consists of a set of forwarding switches in the data plane controlled, managed, and configured by one or more central controllers in the control plane. This is achieved by the controller sending a collection of flow rules to populate the switches flow tables based on the specifications of the *OpenFlow* protocol [35]. The attacker model we assume in the SDN forwarding architecture consists of several switches in the data plane that can execute active attacks on the network traffic and flow rules. The threats imposed by the switches mainly result from an attacker taking full or partial control of the operation of the switches and capable of executing:

Man-In-The-Middle (MITM): and flow poisoning attacks by modifying the flow rules and disrupting the routing operations. This attack jeopardizes the confidentiality and integrity of the network traffic by having the malicious switch(es) pass packets to an attacker-controlled destination switch before forwarding the traffic (which can be maliciously modified) to the legitimate destination. MITM attacks can be detected by inspecting the inbound and outbound traffic flows of the malicious switch(es) to detect any inconsistency in the source and destination packet addresses or by identifying a particular unique address to which the packets are forwarded. A less sophisticated version of the MITM attack is the packet modification attack in which the malicious switch carries out the malicious modification on the network traffic before forwarding it towards the ultimate destination. More details on MITM attacks in SDN contexts can be found in [36].

Denial of Service (DoS) attacks: by injecting illegitimate network traffic. One or more malicious switches can initiate this attack and be coordinated to realize the Distributed DoS (DDoS) attack. Denial of service attacks can be detected by analyzing the outbound flow from the detected malicious switch(es) to inspect large network flows forwarded to a particular victim's destination. More on DoS and DDoS attacks in programmable SDN networks is presented in [37].

Quality of Service (QoS) throttling attacks: by delaying particular network flows. Throttling attacks are a toned-down version of DoS and DDoS attacks and are considered the main cause of malicious SLA violations in cloud systems. QoS throttling attacks can be detected by applying timestamp sketching techniques as proposed in the network security protocol presented in this work or using machine learning classification approaches [38].

Focused or random packet-dropping attacks: to suppress network flows. In these attacks, the malicious switch randomly drops packets to induce a disruption to the forwarding service or selects to drop the packets belonging to a particular source/destination network flow. Probabilistic sketching techniques employed in this work can effectively detect these attacks and isolate the respective malicious switch(es) inducing them. A good reference on these attacks is presented in [39].

Network traffic padding attacks: to cover up malicious traffic dropping [40]. These attacks maliciously conceal the traffic size drop caused by packet-dropping attacks to fool detection mechanisms that operate by comparing the traffic size at the source and destination. A more effective way to mitigate and detect those attacks is by applying content-based comparison schemes provided by probabilistic sketching.

We assume the SDN controller is trustworthy and managed by reliable administration processes. This circle of trust includes all the code modules running on the controller and the flow rules communicated with the data plane. The network forwarding security algorithms are designed to function in the presence of many malicious switches in the data plane. This is considered crucial for ensuring the scalability of the security mechanisms in highly malicious data plane environments, although this comes at the expense of increasing the runtime of the attack localization processes. More details on achieving secure and trustworthy control plane administration and processing in SDN can be found in [41].

V. System model

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

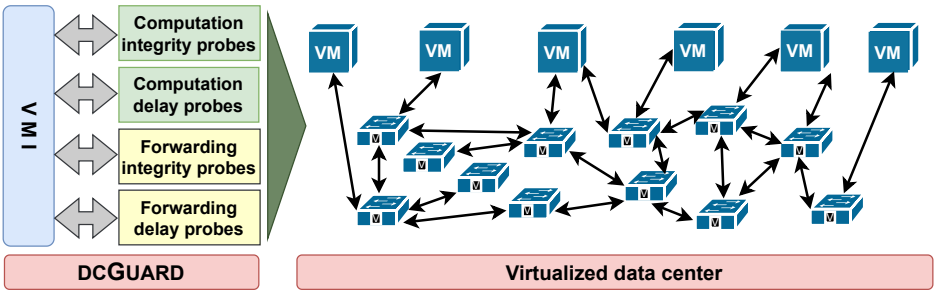

Figure 1: DCGUARD system model.

DCGUARD approach includes two main algorithms to detect malfunctioning nodes that process/forward data by initiating a set of recursive probing mechanisms on the individual tenant VMs and the networking nodes in the data centers. The algorithms are $_P$DCGUARD and $_F$DCGUARD.

A. $_P$DCGUARD *algorithm*

The algorithm verifies the integrity and timeliness of the data processing. The domain of this algorithm is the VMs. The notation used in $_P$DCGUARD is summarized in Table I.

1) Algorithm description: The complete algorithm specification is presented in Algorithm 1. The algorithm represents a function that takes three parameters as input: The tenant to 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 $Ctenant$ VMs execute a particular distributed task. In other words, the algorithm detects and locates any VM that produces invalid computation results or incurs irrational or unacceptable processing delays. It should be mentioned here that the VMs probed are considered untrusted, and the main purpose of the $_P$DCGUARD algorithm is to detect and locate any VM tampering with the processing results or inducing computation delays. The detection process is realized irrespective of the number of misbehaving/malicious VMs or the degree of tampering/delay they are causing above a predefined threshold. 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. Note that the assigned probing tasks are computational procedures based on preconfigured distributed algorithms that require coordination among the VMs to execute and produce a deterministic output. This can range from simple distributed polynomial arithmetic and two-phase commit algorithms to more sophisticated and specialized algorithms such as MapReduce and distributed minimum spanning trees. Without loss of generality, we adopt the distributed matrix multiplication problem based on the divide-and-conquer scheme to apply in the sample testbed implementation. Although this distributed task does not heavily rely on VM coordination, it has well-defined boundaries of the division of labour among the processing VMs, which renders it a viable proof of concept that can be extrapolated to other classes of distributed tasks. The probing system starts by randomly selecting a certain type (Class) of distributed algorithms that comprises several subtasks (Degree) that map to the number of VMs in the probed group (Figure 2 - Step 1). 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 $Ctenant$ to start the probing process is a design choice that needs to be determined. The algorithm can choose a tenant at random to inspect or select one based on a predefined criterion. Without loss of generality, we chose to start with the tenant $Ctenant$ having the largest number of VMs (Figure 2 - Step 2). We believe 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 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 $_P$DCGUARD function on each tenant VM set respectively, as $_P$DCGUARD ($T[i], VM_{Ctenant}(1,\ldots,r)$, "$processing$"), $\forall i = 1,\ldots,k$ and such that $Checked(T[i])$ is false.

As stated in Algorithm 1, the $_P$DCGUARD is a recursive function. It accepts as inputs: (1) the tenant to be currently inspected ($Ctenant$), (2) the VM set of the $Ctenant$, and the type of inspection required, which can be either "processing" or "delay". The base case of the $_P$DCGUARD function is reached when the probed network consists of N remaining VM nodes ($N = 1$ by default), and the algorithm converges to locate the VM node (or set of nodes), resulting in the misbehaving processing or delay of the assigned tasks. (lines 3–5).

In lines 6–10, we select the $Ctenant$ having the maximum number of VMs.

Line 11 saves the probing start time, which aids in calculating the delay in the later phases of the algorithm.

The distributed tasks to be executed on the VMs of the $Ctenant$ 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 (Figure 2 - Step 3). Note that the

Table I: Notations used in the $_P$DCGUARD algorithm.

Symbol	Explanation
$T = \{t_1, \ldots, t_k\}$	The set of k tenants renting VMs in the data center ordered by decreasing number of VMs.
T_{max}	The tenant with the maximum number of VM nodes.
$VM_i = \{v_1^i, \ldots, v_r^i\}$	The set of r VMs belonging to tenant i.
n	The total number of VMs in the data center.
$Class = \{c_1, \ldots, c_p\}$	The set of classes of distributed algorithms that can be used for probing the integrity of VM processing. An example distributed algorithm class used 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 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 indistinguishability of the probing algorithm to be perceived as a normal processing task in the data center.
$D = \{1, \ldots, d\}$	A strictly increasing ordered set of the available size of subtasks comprising the distributed algorithms.
$\Psi_{x,s}$	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.
$\Psi_{x,s}(1, \ldots, s)$	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 [42].
MD	An array of length $k-1$ for storing the hashes (digests) of the output.
TS	An array of length $k-1$ for storing the digest timestamps.
t_{start}	The timestamp indicating the start of task execution on the probed VMs.
$Checked(i)$	A boolean function that returns true if the tenant $i \in T$ is checked for malicious behavior in a probing period.
N	The parameter representing the base case size to stop the recursion. It indicates the number of VMs 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.
$result$	The variable that represents the result of the distributed task by aggregating the output of the subtasks from the various tenant VMs.
$VMI\ Interposition$	The VMI primitive that allows the VMI to execute specific tasks on a particular VM in the data center.
$Conquer$	The function that accumulates the processing subtask output of the distributed task from the tenant VMs and aggregates it into a result variable.
$Output$	The function that computes the result of a subtask execution on a particular tenant VM.
$load$	The function that loads a variable from the VM memory.
$store$	The function that stores a variable in the VM memory.

operation "$(task, input) \longleftrightarrow VM$" indicates that a task with its corresponding input set will be executed on the VM using VMI interposition. (lines 12–15).

The subtask processing results are sent to all other tenants in the data center. More precisely, one randomly selected VM in each tenant group other than the $Ctenant$. (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, and to Figure 2 Steps 4 and 5).

The digest from each other tenant, along with the digest generation timestamp, are respectively stored in the ***MD*** and ***TS*** vectors at index t. (lines 20–25 and Figure 2 Step 6).

The resulting $(k-1)$-element MD is inspected with respect to the expected digest value $H(F(\Psi_{x,s}))$ (Figure 2 Step 7). Suppose $H(F(\Psi_{x,s}))$ does not match any entry in the ***MD*** vector. In that case, this corroborates, with a high confidence level, the fact that the probed $Ctenant$ comprises malicious VM nodes contributing to 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 and Figure 2 Step 8).

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 $Ctenant$ VMs are correctly behaving. Other tenants corresponding to the entries with the non-matching values in the ***MD*** vector are assigned next in a row for probing using the "processing" type argument to the recursive function. (lines 30–31).

Analogously, the algorithm checks any $Ctenant$ 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. (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. (lines 37–39).

Regarding the baseline for misbehavior as far as the computational delay is concerned, this is mainly governed by the SLA QoS terms whose existence in the core specifications of the cloud computing platform is considered a major advantage that we can leverage in designing malicious VM detection algorithms. Employing the SLA terms to establish a baseline of misbehavior renders the malicious detection independent

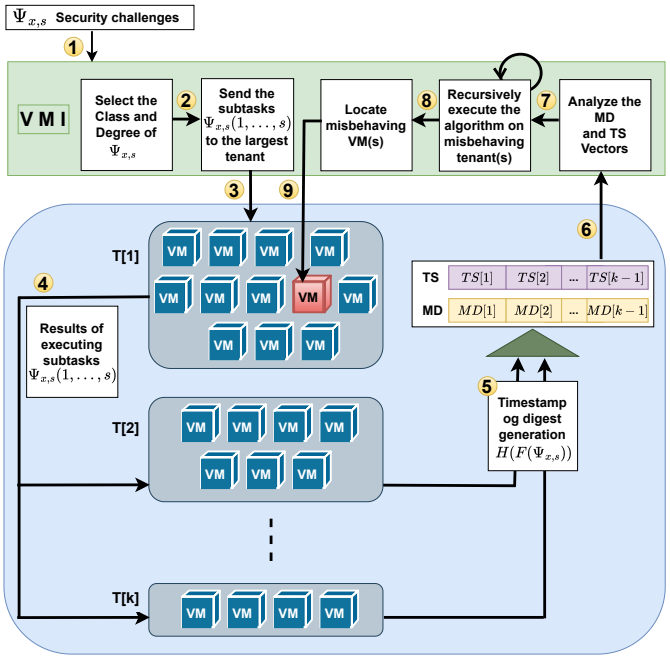

Figure 2: $_P$DCGUARD operations on VMs.

of the individual sources of delay on the individual VMs. No matter what factors produce the delay on the tenant VMs, the detection algorithm will raise an alarm if the delay exceeds the SLA baseline. Any VM domain probed producing an invalid result or an excessive processing delay undergoes the same division process until the recursive base case is reached (Figure 2 Step 9).

2) Algorithmic run-time complexity: In this section, we study the complexity of the $_P$DCGUARD algorithm and focus on three main aspects that are mainly considered in the design of intrusion detection systems: (1) the number of protocol steps to achieve convergence, (2) the computation complexity for executing the distributed probing subtasks on the tenant VMs, and (3) the network traffic size exchanged to achieve the protocol convergence. We compare the complexity of the above 3 aspects to the traditional intrusion detection mechanism that checks each VM independently to detect malicious behavior. A highly significant note that is worth mentioning here is that the $_P$DCGUARD algorithm does not adopt any concurrent inspection on the tenants' VMs when sending the distributed probing subtasks. The main motivation for this design choice is instigated by the nature of task processing in modern virtualized cloud computing architectures where a VM service needs to interact and collaborate with other VM services to achieve a designated computing task. Complying with this computational model has a major impact on the detection capability of the security protocol since many modification/delay attacks are insinuated when a VM interacts with other VMs in the network and may not emanate when the VM is processing the tasks in isolation. For this reason, the distributed $_P$DCGUARD tasks are sent sequentially to and are chosen to spur collaboration between the probed VM and a set of other VMs in the tenant network. Moreover, this gives the intrusion detection module in $_P$DCGUARD major security advantages in detection capacity and accuracy over the traditional detection approach that individually checks each VM in isolation of its virtual operating environment. 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 $_P$DCGUARD 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 producing unreasonable delay in providing the requested services.

Number of recursive protocol steps to achieve convergence: Let $T(n)$ represent the total number of protocol execution steps to achieve convergence in the $_P$DCGUARD 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 run-time classification function, $|VM_k|$ is the number of VMs belonging to the k-th cloud tenant, and f is the function representing the number of protocol recursive steps 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 number of protocol steps of the $_P$DCGUARD 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 $f(|VM_k|)$:

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

It should be noted that this scenario, of having all the VMs in the set $|VM_k|$ malicious, is assumed uncommon in conventional cloud environments, and the $\mathcal{O}(|VM_k|)$ complexity is exactly the number of protocol steps in the traditional VMI algorithm that inspects each VM individually to detect

Algorithm 1: $_P$DCGUARD algorithm

Function $_P$DCGUARD ($Ctenant$, $VM_{Ctenant}(a,\ldots,r)$, *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.: $x = \text{random}(1,\ldots,p)$ 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**
    $_P$DCGUARD $(Ctenant, VM_{Ctenant}(a,\ldots,r/2)$, "*processing*")
    $_P$DCGUARD $(Ctenant, VM_{Ctenant}((r/2)+1,\ldots,r)$, "*processing*")
  else
    for *all* t *where* $MD[t] \neq H(F(\Psi_{x,s})$ *AND* $!Checked(T[t])$ **do**
      $_P$DCGUARD $(t, VM_t(1,\ldots,r)$, "*processing*")
  for *all* t *where* $MD[t] = H(F(\Psi_{x,s})$ **do**
    if *all* $TS[t] - t_{start} \notin \tau$ **then**
      $_P$DCGUARD $(Ctenant, VM_{Ctenant}(a,\ldots,r/2)$, "*delay*")
      $_P$DCGUARD $(Ctenant, VM_{Ctenant}((r/2)+1,\ldots,r)$, "*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**
          $_P$DCGUARD $(t, VM_t(1,\ldots,r)$, "*delay*")

malicious VM behavior. The $\mathcal{O}(\log_2(|VM_k|))$ complexity results in a total complexity $T(n)$ as follows:

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

The logarithmic complexity in the number of protocol steps in $_P$DCGUARD is considered a major improvement over the linear complexity in the traditional inspection approach for large cloud network sizes. A similar analysis for the distrustful scenario of having all VM nodes in the tenant networks exhibiting misbehaving processing or delay patterns results in $T(n) \in \mathcal{O}(n)$ which is the same number of protocol steps complexity realized in the traditional VMI intrusion detection algorithm inspecting the VMs in the data center one at a time.

Computation complexity: Let $C(n)$ be the computational runtime complexity for executing the distributed probing subtasks on the tenants' VMs. We assume that the cost of subtask execution is a constant g compared to the number of VMs in the network. Without loss of generality, we derive $C(n)$ on one cloud tenant with $|VM_k|$ asymptotically approaching the total number of VMs in the network n. The keen reader should be able to extrapolate the analysis to the set of k cloud tenants analogously to what was demonstrated in the previous section on calculating the number of protocol steps $T(n)$. Moreover, we assume the logical cloud scenario where we have a relatively low number of malicious VMs, which results in $T(n) \in \mathcal{O}(\log_2(n))$ as indicated in the previous section. The number of VMs probed in each protocol step starts at n in the first and second steps. Afterward, the number of VMs probed is divided in half till reaching the base case after $\log_2(n)$ iterations. Accordingly:

$$C(n) = g \times (n + n \times (\sum_{i=0}^{(T(n))-1} \left(\frac{1}{2}\right)^i) \quad (5)$$

Solving for the geometric progression in (5) and replacing $T(n)$ with its asymptotic complexity limit $\log_2(n)$ we get:

$$C(n) = g \times (3n - 2) \in \mathcal{O}(n) \tag{6}$$

This is analogous to the complexity of the traditional VMI inspection approach but with a slightly higher constant term. This is justified considering the major security advantage provided by $_P$DCGUARD, which inspects the VM by probing subtasks designed to execute in collaboration with other VMs in the tenant network. This approach complies with the modern cloud services model and provides better detection accuracy than the traditional approach that inspects VMs individually in isolation from their operating environment.

Network Traffic Size: Let $N(n)$ be the complexity function representing the network traffic exchanged to achieve protocol convergence. Deriving $N(n)$ follows a similar procedure as that leveraged in finding $C(n)$ since the VMs probed with the distributed subtasks are those receiving the input signal to execute the subtasks. This results in a linear relationship between $C(n)$ and $N(n)$ having the following form:

$$N(n) = C(n) \times s \times y \tag{7}$$

where s is the input network traffic per VM probe and y is the network traffic exchanged between the probed VM and the other VMs in the tenant network as required by the distributed probing subtask. Both s and y are constants in the size of the network n. From (5) and (7) we get:

$$N(n) = g \times s \times y \times (3n - 2) \in \mathcal{O}(n) \tag{8}$$

Again, compared to the network traffic generated by the traditional VMI approach, $_P$DCGUARD has the same linear complexity but with a larger constant term. This is justified considering the security advantage the $_P$DCGUARD protocol provides and its compliance with the cloud service model when performing the probing mechanism.

B. $_F$DCGUARD *algorithm*

To detect malicious forwarding nodes in the SDN data plane, we leverage the network programming SDN model to divide the network into two equal-degree partitions recursively. This is done by the following mechanisms:

- Apply a graph-theoretic partitioning algorithm to recursively specify two network halves having a minimum number of connecting edges (links).
- Rely on OpenFlow controller messages to isolate the two network partitions specified above.

The network partitions are individually probed with a set of dynamically crafted network packets generated by the SDN controller. More specifically, the SDN controller forks a probe transmitter and receptor daemon processes to send and receive the probing packets respectively. To detect any suspicious activity in the forwarding nodes:

- We apply the Tug-of-War sketching algorithm on the sender and receiver ends to measure the deviation between the sketch generated by the transmitted and received probe packets.
- We employ a Timestamp Summary (TSS) data structure to compute the average delay of the probe packet exchange between the sender and receiver controller processes. The average delay is analyzed to detect any malicious delay induced by misbehaving switching nodes.

Using the above two criteria, the network intrusion detection system can determine the probability of threat instigated by the existence of a misbehaving forwarding unit in the probed network partitions. Using sketching algorithms in the system design dramatically reduces computational complexity when probing the network partitions with a relatively large number of network packets. This aids in improving the accuracy of the intrusion detection mechanisms and enhances the overall system's performance efficiency.

The algorithm verifies the integrity and timeliness of the network transfer. The domain of this algorithm is the network nodes. The $_F$DCGUARD notation is summarized in Table II.

1) Algorithm description: The complete algorithm specification is presented in Algorithm 2.

In lines 1–13, the $_F$DCGUARD algorithm is called on the whole network G_N and operates recursively by calling itself on two subset partitions of the network G_1 and G_2 with minimum interconnecting link edges. The algorithm base case is reached when the network partition size $|G|$ is less than or equal to the granularity of misbehaving forwarding nodes M. In such a case, the algorithm returns the partition containing one or more malicious forwarding nodes. If no malicious partition is detected, then $_F$DCGUARD returns a "valid network" signal.

In lines 15–39, the CHECK-PARTITION procedure generates Tug-of-War sketch summaries of the probing data at the transmit and receive controller daemon processes d_T and d_R, respectively. The probing packet stream $(D : (P_1 \rightarrow P_k))$ is sent from d_T to d_R by traversing all the switches in a given network partition. This is done by the SDN controller disseminating the necessary action rules to ensure that the probing packets traverse each switch in the corresponding network partition. At d_T and d_R, the probing stream $D : (P_1 \rightarrow P_k)$ is input to a Tug-of-War sketching algorithm to generate a compact hash-based representation $T(D_T)$ that is sent to the SDN controller for inspection. This is done as follows: For each probing packet P, a four-wise independent hashing function $4wHash$ is employed, which uniformly maps to a pair of values: an index i in the sketch array and a value result in $[-1, +1]$; the result is added to $T(D)$ at index i. The Tug-of-War sketch of $D : (P_1 \rightarrow P_k)$ is produced as the summation of the dot product of the hashed values and the individual probing packets based on the following equations:

$$(i, result) \leftarrow 4wHash(P) \tag{9}$$

$$T(D_T)[i] \leftarrow T(D_T)[i] + result \tag{10}$$

Based on the linearity property of the Tug-of-War sketch [9], the second norm difference between the two received sketches reflects any variation between the sent and received data streams up to an error e:

$$\delta_{ToW} = |T(D_T) - T(D_R)|^2 \tag{11}$$

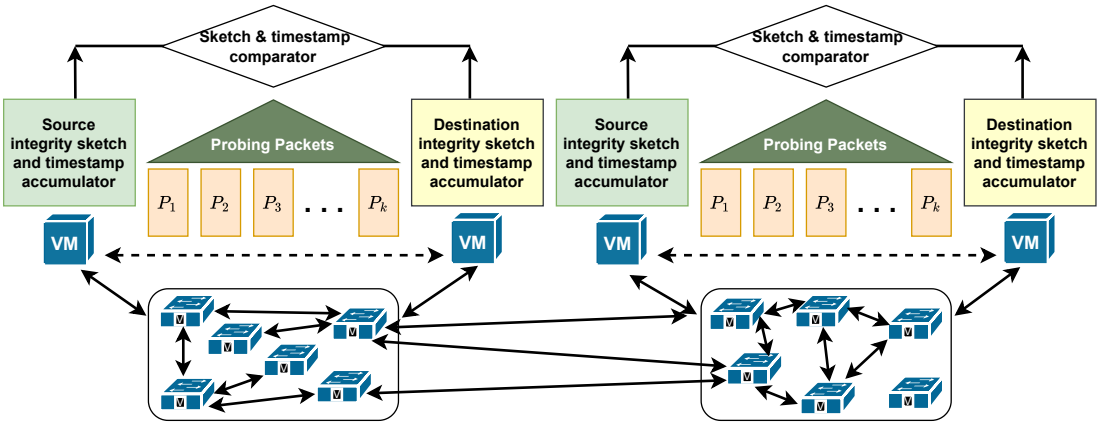

Figure 3: $_F$DCGUARD operations on network nodes.

Table II: Notations used in the $_F$DCGUARD algorithm.

Symbol	Explanation
GN	The graph representing the SDN network.
V	The number of forwarding nodes in the SDN data plane.
E	The number of links in the network.
G	The network partition to be checked for any misbehaving forwarding nodes.
M	The parameter representing the base case size to stop the recursion. It indicates the number of forwarding nodes the algorithm converges to raise a misbehaving output signal. By default, $M = 1$, but the value could be increased to enhance the convergence time to the algorithm at the expense of a less precise misbehaving switch localization. (M is similar to N in the $_P$DCGUARD algorithm).
d_T and d_R	These are, respectively, the transmitting and receiving controller daemon processes responsible for calculating the Tug-of-War sketches and the timestamp summaries.
$4wHash$	A four-wise independent hash function.
$Hash$	A standard collision-resistant cryptographic hash function.
$T(D_T)$	The Tug-of-War sketch of the probing packet stream $(D_T(P_1, \ldots, P_k))$ at the controller transmit daemon process.
$T(D_R)$	The Tug-of-War sketch of the probing packet stream $(D_R(P_1, \ldots, P_k))$ at the controller receive daemon process.
δ_{TSS}	The summation of all the timestamps accumulated between the transmitting and receiving ends.
δ_{ToW}	The second norm difference between the sent and received Tug-of-War sketches subject to an error e.
θ_{TSS}	The average timestamp per probing packet.
τ_{ToW}	A preset threshold above which the deviation in the second norm difference between the transmitted and received Tug-of-War sketches is considered malicious.
τ_{TSS}	A preset threshold above which the difference between the timestamp summary (the average timestamp calculated over all the transmitted and received probe packets) is considered malicious.

The second norm difference calculation follows an efficient linear runtime complexity in the size of the Tug-of-War probabilistic sketches in bytes. This is due to the fact that the second norm operation is realized by summing the squares of the differences between the individual byte elements of the transmitting and receiving sketches on the SDN controller. A more detailed realization of the second norm calculation is demonstrated in the following equation:

$$\delta_{ToW} = \sum_{i=1}^{s} |T(D_T[i]) - T(D_R[i])|^2 \qquad (12)$$

where s is the size of the transmitting and receiving Tug-of-War sketches in bytes. This calculation requires s iterations, rendering its runtime complexity in the order of $\mathcal{O}(s)$. In the sample implementation presented in Section VI-B, we utilized Tug-of-war sketch sizes of 12×10^3 bits = 1500 bytes. The empirical runtime performance of the second norm difference operation on the 2.6 GHz 6-core Intel Core i7 processor was in the order of 380 ms. Tuning δ_{ToW} up to an error e we get:

$$\delta_{ToW} = (1 \pm e) \times \sum_{i=1}^{s} |T(D_T[i]) - T(D_R[i])|^2 \qquad (13)$$

If δ_{ToW} is greater than an empirically set threshold τ_{ToW}, this designates a malicious activity. This probing and sketching procedure is repeated every preset time period t to detect misbehaving switch activity in real time. The Tug-of-War sketches $T(D_T)$ and $T(D_R)$ represent a compact hash representation of the probing data streams, which imposes minimal overhead in the network traffic and storage requirements. Moreover, this allows for an efficient set of mechanisms for detecting a deviation in the relatively large probing packet streams. Like the Tug-of-War sketch generation and computation, the TSS data structures are generated and computed at the d_T and d_R daemon processes concurrently to detect any malicious delay in the forwarding operations. The TSS generation is described as follows: each probe packet P is hashed using a standard collision-resistant hash function to a value j, which is utilized as an index in the TSS_T and TSS_R arrays. These arrays represent the summation of the timestamps of the probing packets that map to the hash value j. The main objective is calculating the average timestamp per probing packet θ_{TSS}. This is done by dividing the summation of all the timestamps between the transmitting and receiving ends, δ_{TSS}, by the

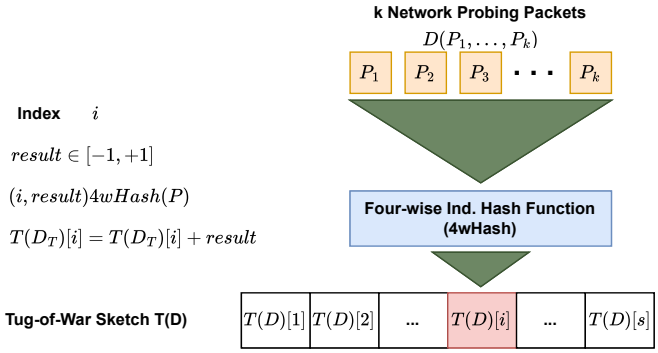

Figure 4: The ToW operations on the probing data stream.

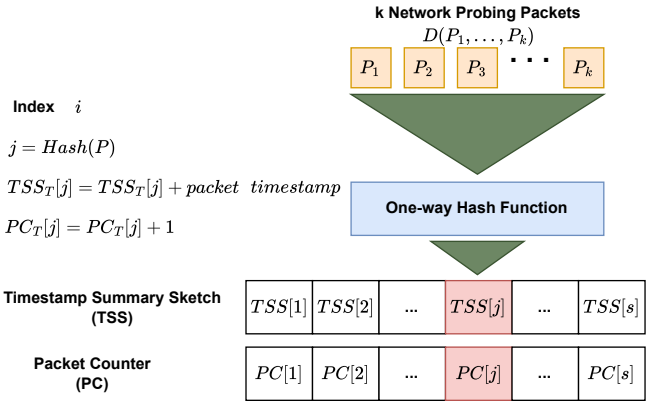

Figure 5: The TSS operations on the probing data stream.

total number of packets, VAL_{TSS}, received correctly on the receiving end. Figures 4 and 5 demonstrate the operation of the Tug-of-War sketch and timestamp summary on the probing data stream.

The CHECK-PARTITION function finally sends $T(D_T)$, TSS_T, and PC_T to the SDN controller for inspection. Analogously, at the receiving probing process, d_R, $T(D_R)$, TSS_R, and PC_R are generated and sent to the SDN controller for comparison with $T(D_T)$, TSS_T, and PC_T for the aim of detecting any malicious activity in the forwarding operation. The SDN controller evaluates the second norm difference δ_{ToW} of the transmit and receiving sketches $T(D_T)$ and $T(D_R)$. If δ_{ToW} is bigger than the threshold τ_{ToW}, the network is reported as malicious in terms of the correctness of the forwarding operations. Similarly, the average transmit and receive packet timestamps difference is evaluated on the probing packet stream $(D : (P_1 \rightarrow P_k))$. This is achieved by checking that the vectors PC_T and PC_R at each index j are equal; this indicates that the corresponding probing packets are correctly received and the timestamp counters at that index j are valid. The timestamp difference $TSS_T[j] - TSS_R[j]$ is evaluated and added to the δ_{TSS}, which represents the summation of all the timestamps accumulated between the transmitting and receiving ends, and the average timestamp per probing packet θ_{TSS} is calculated using the following formula: $\theta_{TSS} = \delta_{TSS}/VAL_{TSS}$

If θ_{TSS} is greater than an empirically set threshold τ_{TSS}, the network is marked as malicious with respect to the timely delivery of packets. This is mainly caused by a malicious forwarding unit producing unreasonable delays.

2) Algorithmic run-time complexity: The average runtime complexity of the $_F$DCGUARD algorithm is realized when a malicious behavior is depicted in one of the network partitions inspected by the intrusion detection system. The algorithmic complexity consists of three main components:

- A recursive cost function represents half the problem size since the algorithm focuses on one of the two network partitions in the current call to the $_F$DCGUARD functions.
- The cost of dividing the network into two subset partitions G_1 and G_2 with minimum interconnecting links. We employ Karger's graph cut algorithm to implement this process, which runs in a time complexity of $\mathcal{O}(V^2 E \log(V))$ on a network $G(V, E)$ [43].
- The cost of the CHECK-PARTITION function, which consists of the probing and sketching mechanism. Since the cost here relies mainly on the Tug-of-War sketch size and not on the number of the forwarding nodes V, this contributes a constant time, say C, to the complexity.

Adding the above-listed cost components, we get the following run-time complexity recursive function:

$$T(V) = T(V/2) + \mathcal{O}(V^2 E \log(V)) + C \tag{14}$$

Solving the recurrence relation in Equation (14) results in an average case run-time complexity:

$$T(V) = \mathcal{O}(V^2 E (\log V)^2) \tag{15}$$

The time complexity presented in (15) is polylogarithmic in the number of forwarding nodes V and the number of network links E. Suppose the intrusion detection was done using the deterministic approach without relying on programmable SDN virtualization functions. In that case, every path of every possible number of forwarding nodes from the probing source to the probing destination has to be checked for malicious/misbehaving forwarding switches. Assuming no forwarding loops in the networks (the application of Spanning Tree protocols can enforce this), the number of paths of size r is the combination $\binom{V}{r}$. Accordingly, the total possible paths of sizes 1 to V is the summation of the combinations:

$$\binom{V}{1} + \binom{V}{2} + \ldots + \binom{V}{V-1} + \binom{V}{V} \tag{16}$$

The runtime complexity to check the above number of paths for malicious/misbehaving switches is an exponential runtime of $\mathcal{O}(2^V)$ since (16) evaluates to 2^V. Achieving a polylogarithmic time complexity using the $_F$DCGUARD approach compared to an exponential runtime with the traditional intrusion detection approach corroborates the efficiency of $_F$DCGUARD and, as a result, its scalability in large-scale SDN networks.

VI. PERFORMANCE EVALUATION

This section presents the performance evaluation for the DCGUARD approach. The main two algorithms of DCGUARD were evaluated separately. The $_P$DCGUARD algorithm was evaluated in two directions: (1) We deployed a stripped-down

Algorithm 2: $_F$DCGUARD algorithm

Function $_F$DCGUARD (G_N, d_T, d_R):
    $_F$DCGUARD (G, d_T, d_R);
    if CHECK-PARTITION(G, d_T, d_R) = *"misbehaving"* **then**
        if $|G| \leq M$ **then**
            return G (G is the network partition containing the malicious forwarding unit)
        else
            Divide network into two subset partitions G_1 and G_2, with minimum interconnecting links
            $_F$DCGUARD (G_1, d_T, d_R)
            $_F$DCGUARD (G_2, d_T, d_R)
    else
        return "valid network"

CHECK-PARTITION(G, d_T, d_R)
- Generate random probing packet stream $D_T(P_1, \ldots, P_k)$ at d_T and transmit to d_T over the network.
- Create the Tug-of-War sketch $T(D_T)$ and the timestamp summary TSS_T at d_T as follows:
for *each packet* $P \in S_1$ **do**
    $(i, result) \leftarrow$ 4wHash(P)
    $T(D_T)[i] \leftarrow T(D_T)[i] + result$
    $j \leftarrow$ Hash(P)
    $TSS_T[j] \leftarrow TSS_T[j] + packettimestamp$
    $PC_T[j] \leftarrow PC_T[j] + 1$
- Transmit $T(D_T)$, TSS_T, and PC_T to SDN controller for inspection
- Analogously compute steps 2 and 3 on d_R to generate $T(D_R)$, TSS_R, and PC_R. Send $T(D_R)$, TSS_R, and PC_R to the SDN controller for inspection, where at the controller:
$\delta_{ToW} = |T(D_T) - T(D_R)|^2$
for i *from* 1 *to* k **do**
    if $PC_T[i] = PC_R[i]$ **then**
        $\delta_{TSS} = \delta_{TSS} + TSS_R[i] - TSS_T[i]$
        $VAL_TSS = VAL_TSS + PC_T[i]$
$\theta_{TSS} = \delta_{TSS}/VAL_{TSS}$
if $\delta_{ToW} \geq \tau_{ToW}$ *OR* $\theta_{TSS} \geq \tau_{TSS}$ **then**
    return "misbehaving"
else
    return "valid"

implementation of the $_P$DCGUARD algorithm on a Microsoft Azure cloud. (2) We implemented a proof-of-concept testbed emulation of the system design on the Mininet network emulator. For the $_P$DCGUARD algorithm, we also evaluated using a proof-of-concept testbed emulation on the Mininet network emulator. Mininet [44] enables the creation of a large-scale 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 SDN modules. The main motivation behind the use of Mininet in emulating the VMI security system is instigated by 3 main aspects: (1) Mininet leverages process-based lightweight virtualization supported by network namespaces where VMs can run real Linux-based OSes with near-native processing speeds on the actual hardware processors. This is corroborated by the realistic figures of the $_P$DCGUARD performance evaluation and the consistent results achieved in the real-world testbed on the Microsoft Azure cloud infrastructure. (2) Mininet supports a 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 access the underlying processing nodes in the virtualized hosts. This is because Mininet's SDN controller is deployed on the local host system running the network virtualization hypervisor with direct access to the VM runtime and memory resources. (3) Mininet innately implements the OpenFlow and simulates SDN controllers and switches supporting this protocol. This was a significant advantage for simulating the $_F$DCGUARD protocol on top of it as well. The VM misbehavior is simulated in Mininet by inducing explicit modification on one or more units in the input-processing-output pipeline. For instance, changing the input fed to the distributed task, the result of an intermediate computation, or the final output result of the distributed task would result in a processing flaw detected by the $_P$DCGUARD algorithm and is considered a malicious attempt to disrupt the computation of the distributed task. We employed a VMware Linux VM to run the Mininet network emulator. The VM is hosted on VMware Fusion Professional Version 12.1.0 and runs on a MacBook Pro, Intel Core i9 @ 2.4GHz, with 64GB of main memory. The guest system was assigned 4 cores and 33.5GB of memory, and it was running on Ubuntu 14.04 (64-bit) using Floodlight v1.2.

A. $_P$DCGUARD

Two ways are used for the performance evaluation:

1) Emulation: The system configuration we followed is comprised of 8 cloud tenants (T1 to T8) leasing several VMs, $\boldsymbol{r}$, as indicated by Table III. This implies a network size of $n = 3875$ VMs, the largest number of virtual hosts we could boot on a single emulation laptop with the above-listed configuration. To emulate the malicious behavior in the

Table III: Samples of VM configurations.

Tenant	T1	T2	T3	T4	T5	T6	T7	T8
No. of VMs *r*	1250	1000	750	500	200	100	50	25

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 all the VM nodes in the network demonstrate malicious behavior. The $_P$DCGUARD algorithm convergence time for each network malicious rate is computed 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. The traditional VMI technique sets the input matrix sizes to 100×100 elements of random values in the $[1, 500000]$ range. In the $_P$DCGUARD algorithm, to ensure a fair comparison with the traditional technique, we selected the parent problem size to ensure that each subtask consists of multiplying input matrices of 100×100 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 $_P$DCGUARD algorithm specifications. The $_P$DCGUARD probing mechanisms are periodically replicated every 10 minutes over 7 days. We selected a random malicious rate ρ from the set $\{0, 5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100\}$. The average run-time for completing a $_P$DCGUARD probing step is calculated for the 12 different malicious rates ρ, and this is respectively compared to the run-time of executing the standard VMI intrusion detection algorithm that checks each VM individually for correct processing and reasonable delay.

The results in Figure 6a remarkably comply with the theoretical analysis presented in the previous section. As expected, when the malicious rate is relatively low, the $_P$DCGUARD algorithm exhibits a protocol step complexity in the order of $\log_2(n)$. As the malicious rate increases above the 50% - 60% mark, the protocol step 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 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 [45]. 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 using the above equations but by replacing $t_{\max}$ by $t_{\min}$ and $\tau_{\max}$ by $\tau_{\min}$.
The gains α and β are set to $\frac{1}{4}$ and $\frac{1}{8}$ respectively [45].

2) Real cloud testbed: We deployed a stripped-down implementation of the $_P$DCGUARD algorithm on a Microsoft Azure virtual network of 16 VMs. We employed the AV2 Series VM profile using the $Standard - A4 - v2$ size configured with a 4-core $Intel\ Xeon$ E5-2673 v3 2.4 GHz processor and 8 GiB RAM. We assume that the 16 VMs on the Microsoft Azure network belong to one cloud tenant, T1. We followed an analogous approach to that in the simulation by inducing intentional processing flaws in a set of VMs, resulting in a malicious rate ρ. We selected the same malicious rate, 12ρ, which we employed in the simulation to determine the number of malicious/non-malicious VMs as indicated in Table IV.

The $_P$DCGUARD algorithm convergence time for each network malicious rate is computed and compared to that of the traditional VMI intrusion detection methodology. Similarly, we employed a set of matrix multiplication distributed tasks to probe the Azure VMs. To get more sensible convergence times on the relatively small number of probed VMs, we utilized matrix multiplication problems comprising input matrix sizes of 1000x1000 elements of random values in the $[1, 500000]$ range. In the $_P$DCGUARD algorithm, we ensured that each subtask consists of probing problems of size 1000x1000 to comply with the problem sizes used in the traditional VMI approach. The probing experiments on the Azure VMs are periodically replicated every 20 minutes over 3 days. We selected a random malicious rate ρ discretely from the set

Table IV: The malicious rate ρ and the corresponding number of malicious nodes.

ρ	0	5	10	20	30	40	50	60	70	80	90	100
Number of non-malicious VMs	16	15	14	13	11	10	8	6	5	3	2	0

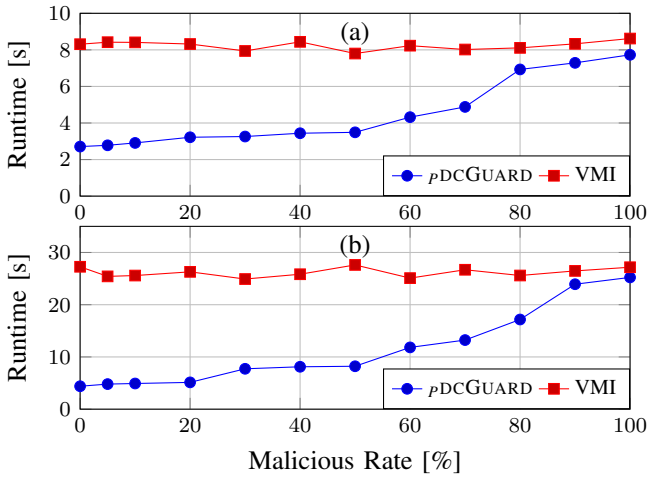

Figure 6: Convergence time in seconds vs. the malicious rate ρ using emulation (a) and a real testbed (b) for FDCGUARD.

$[0, 5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]$. The average convergence time for each of the 12 malicious rates ρ is compared to the runtime of executing the standard VMI intrusion detection algorithm, as indicated previously in the simulation section. The convergence time results for each malicious rate ρ are presented in Figure 6b. The results in Figure 6b corroborate the simulation results, which comply with the theoretical performance analysis. The same convergence time pattern is realized in the real cloud implementation, which increases as the malicious rate ρ increases. The convergence time starts from the order of $\mathcal{O}(\log_2(n))$ runtime complexity for low malicious rates and then gradually increases to asymptotically approach $\mathcal{O}(n)$ for large malicious rates. Again, as demonstrated in the simulation results, the turning point after which the convergence time exhibits the increase above the logarithmic runtime complexity is realized at the 50% - 60% ρ values. After this point, the runtime complexity incrementally approaches the performance of the traditional VMI approach.

B. FDCGUARD

We emulated the FDCGUARD system model on a virtualized network using Mininet. The SDN controller employed is FloodLight 1.2 [46]. The network topology is based on the FatTree architecture. The FatTree topology is chosen due to its popularity in modern data center deployments. The virtual networks selected in the implementation consist of 250 to 2000 forwarding nodes in increments of 250. We realized the Mininet deployment on a MacBook Pro 16" 2019 2.6 GHz 6-core Intel Core i7, 64 GB of 2666 MHz DDR4.

The d_T and d_R probing daemons are implemented as modules on the FloodLight SDN controller. The main role of these modules is to: (1) exchange the probing packet stream $(P1, \ldots, Pk)$, (2) generate $T(D_T), TSS_T, PC_T$ and $T(D_R), TSS_R, PC_R$ on the transmitting and receiving ends respectively and (3) calculate the deviation between the sent and received sketches and timestamp accumulators and decide, based on the threshold values, if a malicious behavior exists in the probed network partition or not. Any malicious behavior detected results in a recursive execution of the algorithm on the currently probed network partition.

Table V: Parameters used in the FDCGUARD Implementation.

Parameter	Value(s)
Network Size	250, 500, 750, 1000, 1250, 1500, and 2000
τ_{ToW}	0.0928
τ_{TSS}	41.428 ms
k	10^6 packets
$\lvert T(D_T)\rvert, \lvert T(D_R)\rvert$	12×10^3 bits
$\lvert TSS_T\rvert, \lvert TSS_R\rvert$	12×10^3 bits
Recursion base case M	3
(*PA*, *PD*)	(5%, 2%), (5%, 5%), (2%, 10%), (5%, 10%), (10%, 2%), (10%, 5%), (10%, 10%)

We introduce an attack vector consisting of two main parameters: (1) *PA*: the percentage of malicious forwarding nodes inducing an active modification attack on the network traffic (packet dropping, packet interception, packet injection, and packet alteration), and (2) *PD*: the percentage of malicious forwarding nodes imposing unacceptable delays in the packet forwarding process. The malicious forwarding nodes executing modification and delay attacks are chosen randomly throughout the experiments. Table V presents the parameters used in the implementation, including the threshold values for detecting active modification and delay attacks. For each network size V and (*PA*, *PD*) pair, we executed 10 replicated experiments to calculate the average number of recursive steps and the average protocol convergence time needed to detect the malicious forwarding nodes. Figure 7 shows the results realized. Apparently, the number of recursive steps and the time needed for the FDCGUARD protocol convergence is directly proportional to the network size and the percentage of malicious switches imposing packet alteration and delay attacks represented in the malicious percentage pair (PA, PD). Based on the results above, an interesting observation is that the FDCGUARD algorithm demonstrates a close number of recursive steps for relatively small network sizes per (PA, PD) pair. The difference in the number of recursive steps increases for network sizes above 750 forwarding nodes. The number of recursive steps is in the order of 30 steps for a 250 forwarding switches network and respectively increases to the order of 50 and 70 steps for 500 and 750 forwarding nodes' networks. For network sizes above 750 switches, the FDCGUARD algorithm starts exhibiting an increase in the number of recursive steps with the network size and the percentage of misbehaving switches represented in the pair

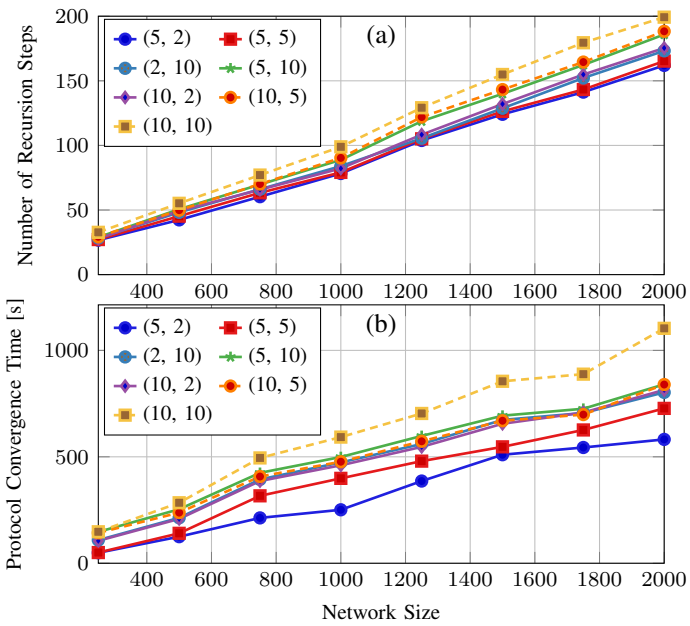

Figure 7: Number of recursive steps (a) and convergence time (b) for various (PA, PD) malicious pairs in $_F$DCGUARD.

Table VI: The τ_{ToW} and τ_{TSS} threshold values.

TCA	τ_{ToW}	τ_{TSS}
1	6.4	57.5
2	3.2	55
3	1.6	52.5
4	0.8	50
5	0.4	47.5
6	0.2	45
7	0.1	42.5
8	0.05	40
9	0.025	37.5
10	0.0125	35
11	0.00625	32.5
12	0.003125	30

Table VII: Optimal threshold values based on ROC analysis.

Network Size	(TCA, TCD)	τ_{ToW}	τ_{TSS}
250	$(7, 8)$	0.1	40
500	$(8, 8)$	0.05	40
750	$(8, 7)$	0.05	42.5
1000	$(7, 7)$	0.1	42.5
1250	$(6, 7)$	0.2	42.5
1500	$(7, 7)$	0.1	42.5
2000	$(8, 8)$	0.05	40

(PA, PD). The difference in the number of recursive steps between the highest malicious pair $(PA, PD) = (10, 10)$ and the smallest $(PA, PD) = (5, 2)$ increases from 20 steps in network sizes of 1000 forwarding nodes to almost 40 recursive steps when increasing the network size to 2000 forwarding nodes ($\cong$ 30 steps for 1250 and 1500 network sizes).

The convergence time is highly sensitive to the rate of misbehavior in the network and is even more highly sensitive to the increase in the number of forwarding nodes in the network. The convergence time increases from 49.32 sec for $(PA, PD) = (5, 2)$ to 147.84 sec for $(PA, PD) = (10, 10)$ ($\cong$ 66% increase) for the smallest network size of 250 forwarding nodes. For the largest network size of 2000 forwarding nodes, the convergence time increases from 581.76 sec for $(PA, PD) = (5, 2)$ to 994.92 sec for $(PA, PD) = (10, 10)$ ($\cong$ 42% increase). The rate of increase of the convergence time from the smallest network size of 250 switches to the largest network size of 2000 switches exceeds 82% for all the (PA, PD) pairs with the $(2, 10)$ pair touching the 93% increase in convergence time. This can be justified by the relatively high malicious delay (PD), which is the main source contributing to the increase in the convergence time.

Modern intrusion detection is based on threshold analysis and configuration. The Threshold τ_{ToW} is tested against the second norm difference δ_{ToW} between the sent and received Tug-of-War sketches subject to an error e. δ_{ToW} values greater than τ_{ToW} indicates a malicious activity since this is an indication that the sent and received traffic differs in unacceptable values with respect to the threshold τ_{ToW}. A very important step in analyzing the performance and effectiveness of an intrusion detection protocol is tuning the threshold parameters to satisfy the security requirements of the protocol with a high confidence level. We relied on the ROC analysis to achieve this. ROC analysis provides a better estimation of the protocol threshold values to realize a balance between the successful attack detection of the intrusion detection algorithm and the false positive alarms generated. For this purpose, we created two threshold configurations: (1) The Active Modification Attack Threshold Configuration (TCA) which is responsible for tuning the τ_{ToW} threshold, and (2) the Delay Threshold Configuration (TCD) which is responsible for tuning the τ_{TSS} threshold. For each configuration, we indexed 12 values of the respective thresholds as indicated in Table VI. In TCA, τ_{ToW} starts with 6.4 at index 1 to reach 0.003125 at index 12 (0.5 multiples). Analogously, in TCD, τ_{TSS} starts with 57.5 at index 1 to reach 30 at index 12 (2.5 ms increments).

For finding the optimal τ_{ToW} and τ_{TSS} threshold values, we simulated 300 active modification and delay attacks on the network sizes indicated in Table V. For each network size, we plot the ROC curves to find the optimum TCA and TCD indices providing an acceptable balance between the false positives and true positives probabilities when running the $_F$DCGUARD algorithm. The TCA/TCD ROC curves for each respective network size are presented in Figure 8.

Analyzing the ROC results, we collected the optimal point for each network size that best balances the false positives and true positives generated. The optimal (TCA, TCD) points and their respective τ_{ToW} and τ_{TSS} threshold values are recorded in Table VII. We took the average of the optimal τ_{ToW} threshold and τ_{TSS} threshold values in the implementation.

VII. CONCLUSION

This work presents DCGUARD, a security approach for securing processing and forwarding operations in virtualized cloud data centers. The main contribution of this work is represented in employing the VMI interposition and introspection functions to detect and isolate any source of maliciousness in the tenants' VMs. SDN virtualization functions and Probabilistic sketching techniques are adopted to detect malicious

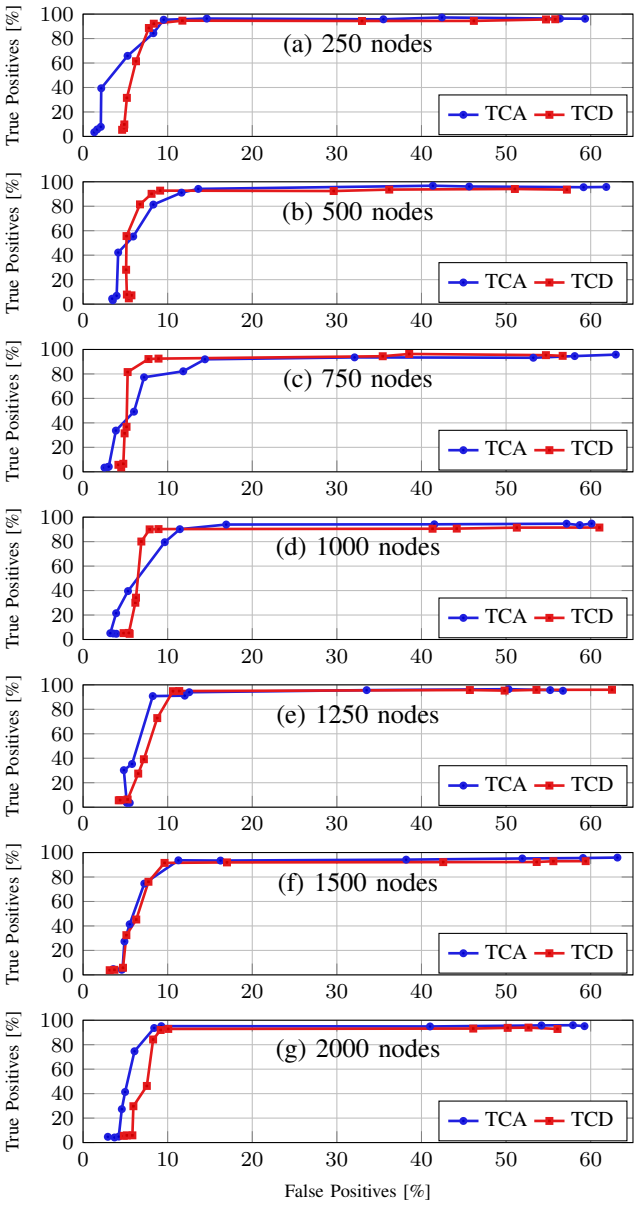

Figure 8: ROC curves for the TCA and TCD configurations for different network sizes.

forwarding operations among networking nodes. DCGUARD follows a divide-and-conquer strategy, resulting in improved protocol efficiency, more accurate detection security, and better compliance with the modern cloud service model. The computational complexity and network traffic size are comparable to the traditional approach (linear time complexity) but with a major security advantage and better compliance with the modern cloud service model. The system is implemented in a virtualized cloud environment using the Microsoft Azure cloud and the Mininet network emulator. Experimental performance measurements corroborate the advantage of DCGUARD over the traditional VMI intrusion detection strategy.

REFERENCES

[1] A. Al-Dulaimy, M. Jansen, B. Johansson, A. Trivedi, A. Iosup, M. Ashjaei, A. Galletta, D. Kimovski, R. Prodan, K. Tserpes *et al.*, "The computing continuum: From iot to the cloud," *Internet of Things*, vol. 27, p. 101272, 2024.

[2] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection and monitoring through vmm-based "out-of-the-box" semantic view reconstruction," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 2, pp. 1–28, 2010.

[3] M. Shamseddine, A. Al-Dulaimy, W. Itani, T. Nolte, and A. V. Papadopoulos, "Nodeguard: A virtualized introspection security approach for the modern cloud data center," in *22nd IEEE Int. Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2022, pp. 790–797.

[4] A. Al-Dulaimy, W. Itani, J. Taheri, and M. Shamseddine, "bwslicer: A bandwidth slicing framework for cloud data centers," *Future Generation Computer Systems*, vol. 112, pp. 767–784, 2020.

[5] 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.

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

[7] H. wook Baek, A. Srivastava, and J. Van der Merwe, "Cloudvmi: Virtual machine introspection as a cloud service," in *2014 IEEE International Conference on Cloud Engineering*. IEEE, 2014, pp. 153–158.

[8] E. Haleplidis, K. Pentikousis, S. Denazis, J. H. Salim, D. Meyer, and O. Koufopavlou, "Software-defined networking (sdn): Layers and architecture terminology," Tech. Rep., 2015.

[9] G. Cormode and M. Garofalakis, "Sketching streams through the net: Distributed approximate query tracking," in *Proceedings of the 31st international conference on Very large data bases*, 2005, pp. 13–24.

[10] R. R. Kompella, K. Levchenko, A. C. Snoeren, and G. Varghese, "Every microsecond counts: tracking fine-grain latencies with a lossy difference aggregator," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 4, pp. 255–266, 2009.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

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

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

[20] X. He and R. Li, "Malware detection for container runtime based on virtual machine introspection," *The Journal of Supercomputing*, vol. 80, no. 6, pp. 7245–7268, 2024.

[21] 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.

[22] Z. Fan, Y. Xiao, A. Nayak, and C. Tan, "An improved network security situation assessment approach in software defined networks," *Peer-to-Peer Networking and Applications*, vol. 12, no. 2, pp. 295–309, 2019.

[23] S. Shin and G. Gu, "Cloudwatcher: Network security monitoring using openflow in dynamic cloud networks (or: How to provide security monitoring as a service in clouds?)," in *2012 20th IEEE international conference on network protocols (ICNP)*. IEEE, 2012, pp. 1–6.

[24] S. Shin, H. Wang, and G. Gu, "A first step toward network security virtualization: From concept to prototype," *IEEE Transactions on Information Forensics and Security*, vol. 10, no. 10, pp. 2236–2249, 2015.

[25] Y. Wang, Y. Zhang, V. Singh, C. Lumezanu, and G. Jiang, "Netfuse: Short-circuiting traffic surges in the cloud," in *2013 IEEE international conference on communications (ICC)*. IEEE, 2013, pp. 3514–3518.

[26] H. Moraes, R. Nunes, and D. Guedes, "Dcportalsng: Efficient isolation of tenant networks in virtualized datacenters," *Proc. 13th ICN*, pp. 230–235, 2014.

[27] T. A. Tang, L. Mhamdi, D. McLernon, S. A. R. Zaidi, and M. Ghogho, "Deep recurrent neural network for intrusion detection in sdn-based networks," in *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*. IEEE, 2018, pp. 202–206.

[28] S. BOUKRIA and M. GUERROUMI, "Intrusion detection system for sdn network using deep learning approach," in *2019 International Conference on Theoretical and Applicative Aspects of Computer Science (ICTAACS)*, vol. 1. IEEE, 2019, pp. 1–6.

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

[30] 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.

[31] L. Cosseron, M. Quinson, L. Rilling, and M. Simonin, "Hiding virtual machine introspection pauses in networked sandboxes with network simulation," Ph.D. dissertation, Inria Rennes-Bretagne Atlantique & IRISA, 2023.

[32] D. Migault, M. A. Simplicio, B. M. Barros, M. Pourzandi, T. R. Almeida, E. R. Andrade, and T. C. Carvalho, "A framework for enabling security services collaboration across multiple domains," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2017, pp. 999–1010.

[33] S. Simpson, S. N. Shirazi, A. Marnerides, S. Jouet, D. Pezaros, and D. Hutchison, "An inter-domain collaboration scheme to remedy ddos attacks in computer networks," *IEEE Transactions on Network and Service Management*, vol. 15, no. 3, pp. 879–893, 2018.

[34] Q. Li, H. Huang, R. Li, J. Lv, Z. Yuan, L. Ma, Y. Han, and Y. Jiang, "A comprehensive survey on ddos defense systems: New trends and challenges," *Computer Networks*, p. 109895, 2023.

[35] A. Lara, A. Kolasani, and B. Ramamurthy, "Network innovation using openflow: A survey," *IEEE communications surveys & tutorials*, vol. 16, no. 1, pp. 493–512, 2013.

[36] R. Gonzaga and P. N. M. Sampaio, "Mitigating man in the middle attacks within context-based sdns," in *8th int. workshop on ADVANCEs in ICT infrastructures and services (ADVANCE 2020)*, 2020, pp. 1–8.

[37] B. L. Dalmazo, J. A. Marques, L. R. Costa, M. S. Bonfim, R. N. Carvalho, A. S. da Silva, S. Fernandes, J. L. Bordim, E. Alchieri, A. Schaeffer-Filho *et al.*, "A systematic review on distributed denial of service attack defense mechanisms in programmable networks," *Int. Journal of Network Management*, vol. 31, no. 6, p. e2163, 2021.

[38] S. V. Ramani and R. H. Jhaveri, "Sdn framework for mitigating time-based delay attack," *Journal of Circuits, Systems and Computers*, vol. 31, no. 15, p. 2250264, 2022.

[39] Q. Li, X. Zou, Q. Huang, J. Zheng, and P. P. C. Lee, "Dynamic packet forwarding verification in sdn," *IEEE Transactions on Dependable and Secure Computing*, vol. 16, no. 6, pp. 915–929, 2019.

[40] Z. Xu, H. Khan, and R. Muresan, "Tmorph: A traffic morphing framework to test network defenses against adversarial attacks," in *Int. Conference on Information Networking (ICOIN)*, 2022, pp. 18–23.

[41] S. Y. Zhu, S. Scott-Hayward, L. Jacquin, and R. Hill, *Guide to Security in SDN and NFV: Challenges, Opportunities, and Applications*. Springer, 2017.

[42] 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.

[43] D. R. Karger, "Global min-cuts in rnc, and other ramifications of a simple min-cut algorithm." in *Soda*, vol. 93, 1993, pp. 21–30.

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

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

[46] "Project floodlight," https://github.com/floodlight/, 2022.

Wassim Itani (M'03) received his B.E. in electrical engineering, with distinction, from the Beirut Arab University (BAU) in 2001, his M.E. in computer and communications engineering from the American University of Beirut (AUB) in 2003, and his Ph.D. in electrical and computer engineering from AUB in 2012. Currently, he is an Associate Professor in the Computer Science department at George Mason University, USA. Wassim's research interests include cloud computing trust and security protocols, wireless and body sensor network security and privacy, and the performance evaluation of cryptographic protocols.

Maha Shamseddine (M'03) is an Assistant Professor of computer science at George Mason University, USA. She received her bachelor's degree in electrical engineering from Beirut Arab University in 2001 and her Master's degree in computer and communications engineering from the American University of Beirut (AUB) in 2003. Maha holds a PhD in electrical and computer engineering from AUB in 2017. Maha's research interests include security in virtualized SDN Networks and the application of machine learning models in the fields of networking.

Auday Al-Dulaimy (M'18) received the BSc and MSc degrees in computer science from Al-Nahrain University (Iraq) in 2000 and 2003 respectively, and the PhD degree in computer science from Beirut Arab University (Lebanon) in 2017. He is an Assistant Professor at Mälardalen University and Dalarna University, Sweden. He was a Postdoctoral Researcher at Karlstad University (Sweden) and Mälardalen University (Sweden) from 2018 to 2022, and was a Visiting Researcher at The University of Messina (Italy) in 2023. Auday is an Associate Editor of the Internet of Things journal of Elsevier. His research interests include distributed systems, cloud computing, and edge computing.

Thomas Nolte (M'01–SM'11) was awarded a Ph.D. degree in Computer Engineering from Mälardalen University (MDU), Sweden, in 2006. He was a Visiting Researcher at the University of California, Irvine (UCI), USA, in 2002, and a Visiting Researcher at the University of Catania, Italy, in 2005. He was a Postdoctoral Researcher at the University of Catania in 2006, and at MDU in 2006-2007. Thomas Nolte is a Full Professor of Computer Science at MDU since 2012, a Director of the industrial PhD school Automation Region Research Academy (ARRAY) since 2017, a Research Leader of research in Electrical and Computer Engineering at MDU since 2022, and a Scientific Advisor at ABB since 2012 (2012-2016 @ ABB Corporate Research, and since 2017 @ ABB Robotics).

Alessandro V. Papadopoulos (SM'19) is a Full Professor of Electrical and Computer Engineering at Mälardalen University, Sweden, and a QUALIFICA Fellow at the University of Málaga, Spain. Since March 2024, he has been the scientific leader of Applied AI at Mälardalen University. He received his B.Sc. and M.Sc. (summa cum laude) degrees in Computer Engineering from the Politecnico di Milano, Milan, Italy, and his Ph.D. (Hons.) degree in Information Technology from the Politecnico di Milano, in 2013. He was a Postdoctoral researcher at the Department of Automatic Control, Lund, Sweden (2014-2016) and Politecnico di Milano, Milan, Italy (2016). His research interests include robotics, control theory, real-time systems, and autonomic computing.