USING FORMAL METHODS TOWARDS IMPROVING CLOUD IAAS ENVIRONMENTS

by

Saeed Al-Haj

A dissertation submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in
Computing and Information Systems

Charlotte

2016

Approved by:

Dr. William J. Tolone

Dr. Anita Raja

Dr. Weichao Wang

Dr. James Conrad

ABSTRACT

SAEED AL-HAJ. Using formal methods towards improving cloud IaaS environments. (Under the direction of DR. WILLIAM J. TOLONE)

Cloud computing has become a prominent technology with the potential for tremendously positive effects on the future of computer networks and services. However, as the resources are shared in cloud environments, cloud security is a major concern. As such, for cloud computing to reach its full potential, better security solutions are required (particularly solutions to security issues that are unique and fundamental to the cloud environment). In this dissertation, we present a formal method-based approach to making clouds environments more secure and manageable. The scope of our work addresses one of the three major types of cloud environments, Infrastructure as a Service (IaaS) cloud environments, and is grounded in a common set of formal methods (i.e., binary decision diagrams, constraint satisfaction problems, and computational tree logic). First, we present a formal method-based, semi-automated framework for access control list (ACL) generation to improve IaaS security manageability. Second, we present a formal method-based cloud resource allocation framework that factors in customer security requirements, specifically reachability and ACLs, at the time of virtual machine provisioning. Third, we present a formal method-based framework for virtual machine migration planning, in which a safe migration order is determined to ensure the preservation of security requirement during migration. Fourth, we present a formal method-based framework for virtual machine post migration reconfiguration and verification. Fifth, we provide a formal method-based framework that detects and resolves policy misconfigurations in Software Defined Networks (SDNs), an important, emerging approach to managing cloud computing infrastructures. These frameworks use a common formal method-based approach and were evaluated in simulated environments for their effect on IaaS security. The results demonstrate the efficiency and

usability of these frameworks to improve IaaS security and suggest promising further areas of research.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ACL Access Control List

BDD Binary Decision Diagram

CSP Constraint Satisfaction Problem

CTL Computational Tree Logic

IaaS Infrastructure as a Service

IDS Intrusion Detection System

IPSec Internet Protocol Security

PaaS Platform as a Service

PM Physical Machine

QoS Quality of Service

SaaS Software as a Service

SDN Software Defined Network

SLA Service Level Agreement

SMT Satisfiability Modulo Theories

VLAN Virtual Local Area Network

VM Virtual Machine

VPN Virtual Private Network

CHAPTER 1: INTRODUCTION

In the recent years, cloud computing has become a prominent technology with the potential for tremendously positive effects on the future of computer networks and services. There are multiple factors that make cloud computing attractive. These factors include, but are not limited to: scalability, efficiency, accessibility and the ease of software integration. First, utilizing a pay-as-you-use scheme, cloud computing supports the rapid scaling of computing solutions. Adding of virtual machines can occur much faster than the purchasing of new physical machines. Second, users can setup and/or (re)configure a fully functioning computing solution more quickly and efficiently in a cloud environment as compared to the level of effort required to accomplish the same task within a traditional enterprise network. Moreover, hardware management and support are left to the cloud provider; thereby reducing overall costs to cloud users through economies of scale. Third, data and services deployed in the cloud have greater accessibility. Typically, cloud users require only Internet access to be able to access their data and services. Furthermore, cloud computing not only eases access to data and services, it also provides an efficient solution to system backup and recovery (i.e., virtual machines provide a more flexible solution than physical machines when addressing the issue of availability). Last but not least is the ease of software integration. Cloud providers provide flexible integration platforms for their customers to exchange data between applications. In other words, cloud customers do not need to worry about integrating different business solutions and the exchange of data across applications; it is up to cloud providers to solve these issues. Oracle Fusion middleware [1] and IBM Websphere [2] are examples of platforms for linking applications and services hosted in clouds.

These characteristics of cloud computing, although conceptually simple, are driving a major paradigm shift in the way computing infrastructures are designed and computing services are delivered.

There are three primary types of cloud computing solutions offered by cloud providers: Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS) and Infrastructure-as-a-Service (IaaS). The research reported here focuses on security challenges associated with IaaS clouds. Amazon, Microsoft, VMWare, Rackspace, IBM, Red Hat, and several other vendors have IaaS cloud offerings. For IaaS clouds, the cloud provider owns the servers, storage media, and networking infrastructure, and is responsible for their operation and maintenance. The cloud customer is often charged by the provider on a pay-as-you-use basis for cloud resources. Examples of cloud resources include computing units, storage, operating systems, and networks.

Provisioning is the process of assigning cloud resources to an IaaS (realized as a suite of one or more virtual machines). Once an IaaS is provisioned, cloud resources are used by customers to run applications within an IaaS. Cloud computing utilizes virtualization technology to provide resources to each IaaS. Cloud resources are shared among IaaS instances and these resources are accessed over the network.

As the resources are shared in cloud environments, cloud security is a major concern. Violation of security requirements may jeopardize the entire cloud environment (i.e., all IaaS instances) rather than a single IaaS. Such security concerns make users reluctant to utilize cloud environments, thus, missing out on the projected benefits of cloud computing (i.e., scalability, efficiency, accessibility and the ease of software integration). As such, for cloud computing to reach its full potential, better security solutions are required – particularly solutions to security issues that are unique and fundamental to the cloud environment.

Cloud environments are different from traditional enterprise networks in many aspects: boundaries are less rigid; resource allocation is more dynamic; and, there is

an additional control structure (the cloud infrastructure itself). Each difference is described in greater detail below. First, on traditional enterprise networks, there is a clear boundary between the inside network and the outside network – usually delimited by a DMZ. To secure the inside network, the trusted practice is to deploy security middle-boxes at network edges and entry points. In cloud environments, however, the concepts of inside network and outside network do not apply; the "outside" of an IaaS is still inside the cloud environment. Thus, there is no clear physical separation between the secure and insecure worlds. As such, the potential exists for every virtual machine to reach – e.g., communicate, interact, access – any other virtual machine within the cloud. Second, the dynamic nature of clouds makes cloud resource allocation (i.e., provisioning) subject to change. As a result of this dynamic nature, virtual machines no longer have fixed locations. They can migrate from one physical host to another physical host within the cloud. Moreover, migrating virtual machines requires updating network configurations according to the new state. Third, cloud users have no control over the cloud infrastructure (and the migrations of virtual machines that may occur). They cannot directly access the physical cloud infrastructure. The only way users can access cloud resources is via the Internet. Moreover, cloud users do not participate in the design of the cloud network topology and the placement of security middle-boxes. Although limiting physical access to the cloud infrastructure and hiding the cloud network topology can provide security advantages, they create doubts for users about how serious their security requirements are considered. These differences between cloud environments and traditional enterprise networks introduce new security threats that must be addressed for cloud computing to reach its full potential.

There are many challenges facing cloud computing evolution. The challenges vary from a cloud user perspective to a cloud provider perspective. These challenges include, but are not limited to, manageability, configuration analytics, and security.

The unique characteristics of cloud environments require different approaches to these challenges than traditional enterprise networks.

For the previously mentioned reasons, cloud security problems have to be addressed carefully. We believe that the security solutions for clouds will be non-traditional.

In this dissertation, we present a formal method-based approach to making cloud environments more secure and manageable. The presented approach improves cloud security and manageability along three dimensions. First, we present a cloud resource allocation framework that factors in customer security requirements, specifically reachability and Access Control Lists (ACLs) for cloud virtual machines. Second, we present a framework for virtual machine migration planning, in which a safe migration order is determined to ensure the preservation of security requirements during migration. Also, we present a framework for virtual machine post migration verification and reconfiguration, in which cloud network configuration is verified after migration and reconfigured if needed. Third, cloud networks are shifting towards utilizing Software Defined Networks (SDNs) [3]. SDNs provide a centralized, global view of the network. Also, SDNs offer management support to virtual environments; which make a great benefit for clouds infrastructures. However, like any other network type, SDNs are susceptible to misconfiguration. Towards solving misconfigurations in SDNs, we include a framework within our formal method-based approach that detects and resolves policy misconfigurations. Each of dimension of our approach (resource allocation, virtual machine migration, and SDN policy misconfiguration detection and resolution) while improving cloud security and manageability is grounded a common formal method-based approach. In the following, we provide a high level overview of our approach.

In our approach, cloud users are encouraged to play an active role in defining the security requirements for their virtual machines. This collaboration between the cloud provider and the cloud users can help to bridge the trust gap and make user

involvement in security decisions more practical. When requesting a new set of virtual machines from the cloud provider, the user provides information about the security needs to be fulfilled. The user declares which virtual machines need traffic inspection, traffic encryption/decryption, and anti-collocation requirements. These needs are augmented with the network reachability requirements between all of the user's virtual machines to define the underlying IaaS instance. With our approach, virtual machines for a specified IaaS instance are organized into several groups according to the similarity of their reachability requirements. The resulting groups are used to synthesize access control policies for all virtual machines to reduce the possibility for policy misconfiguration as well as human errors. The formal method-based approach to create access control policies removes the burden from the user and provides a semi-automated tool for simplifying the management of security groups. After grouping all virtual machines logically into security groups, a virtual machine reachability matrix is updated to reflect any new possible communication due to the grouping process. This matrix is used to calculate a security risk measure for each virtual machine. By combining all user inputs – security needs and reachability requirements – with virtual machines specifications – memory, CPU, storage, and bandwidth – the cloud resources are provisioned using a formal method-based approach to run these virtual machines.

After provisioning resources to virtual machines, the cloud provider may migrate virtual machines across physical systems for more efficient resource utilization. The process of migrating virtual machines requires that all security and capacity constraints be maintained during the intermediate transition states. In other words, a proper migration plan is required to perform safe virtual machine migration. A proper migration plan is not only necessary for safe transitioning, but also necessary for updating network configurations properly due to migration changes. Here again, we provide a formal method-based approach to developing a proper migration

plan. After developing a migration plan, it is important to verify that cloud network configuration is preserved after migration. The framework compares cloud network configurations before and after VM migration. The goal is to have an equivalent configuration for the new state after migration to the old state before migration. We provide a formal method-based approach to verify the cloud network configuration after performing the migration.

The SDN architecture provides a crucial support for cloud infrastructure by providing a central and global view that make cloud management easier. The functionality of a SDN depends on the proper configuration of its policy that is implemented in a the controller and switch flowtables. The presence of policy misconfigurations affects the behavior of the SDN and jeopardizes its security. We provide a formal method-based approach to detect misconfigurations in switch flowtables.

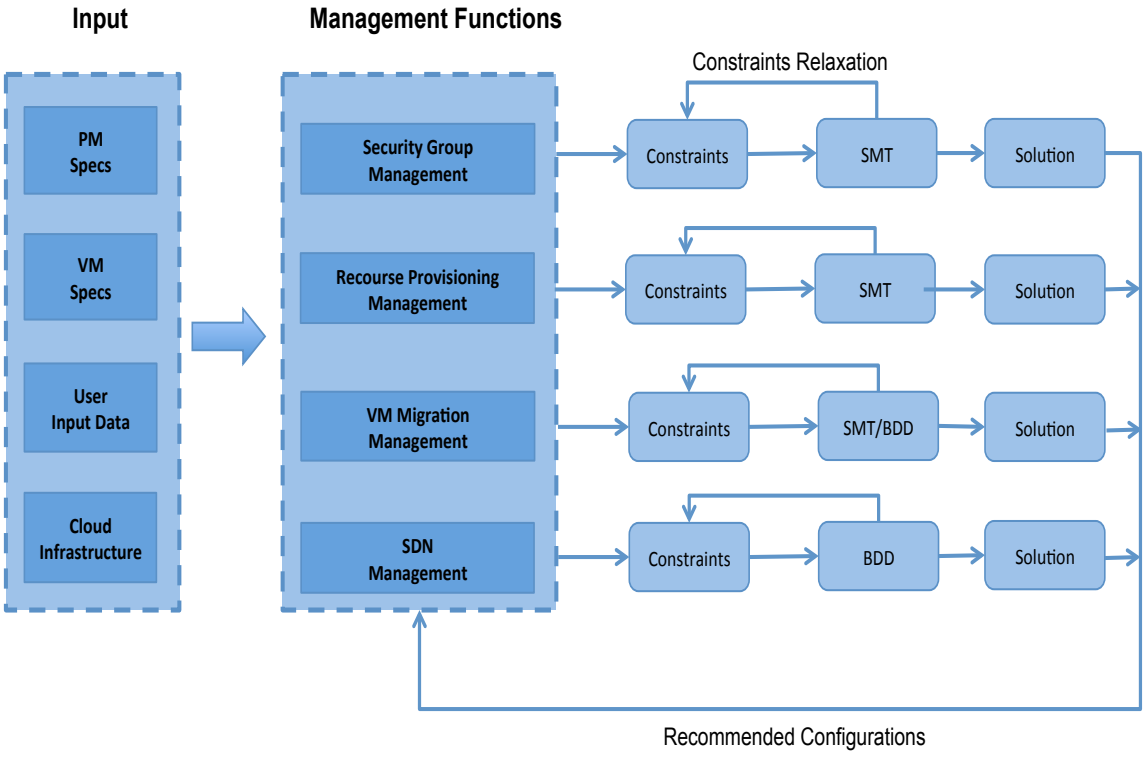

Figure 1.1: The overall methodology

Figure 1.1 shows the overall methodology. All formal frameworks are grouped

into one formal method-based approach. Each dimension of the security function is assigned a management function. There are four management functions: security group management, resource provisioning management, VM migration management, and SDN management. The inputs to the system include: Physical Machine (PM) specifications, VM specifications, user input data, and cloud infrastructure information. All inputs are encoded as constraints which in turn are fed to the Satisfiability Modulo Theories (SMT) solver. The SMT solver will find a satisfiable solution to all given problems and send as configurations recommendations to the SDN management. If there is no satisfiable solution, some constraints will be relaxed to give more flexibility to the SMT solver to find a satisfiable solution.

Next, we provide a more specific introduction to each security/manageability problem and outline the proposed solution to overcome its challenges.

1.1 Managing Access Control Policies for Virtual Machines

Within IaaS instances, each VM requires an ACL to manage network traffic from/to that VM. Managing a single ACL for each VM is an error-prone task, especially as the number of VMs increases [4]. To mitigate this risk, current cloud providers allow VMs to be grouped together based on their incoming traffic similarity. All VMs in such a group share a common ACL. Yet, it is the user's responsibility to group similar VMs and to configure VM access control lists. Both tasks are subject to human error. Another limitation in the current approaches is the number of security groups that a user can use and the size of the ACL for each group (i.e., the number of rules in a single ACL). For instance, Amazon EC2-classic limits the number of security groups to 500 and the number of rules that can be added to a security group to 100, while in Amazon EC2-VPN the number of security groups and the number of rules are 5 and 50, respectively [5].

Whether there is a maximum of 250 or 50,000 rules, the possibility for human error remains a concern. As such, a framework and semi-automated tool set for managing

and creating groups for IaaS VMs can reduce security configuration errors. In particular, such a framework can reduce the chances of having ACL misconfigurations [6]. Under such a framework, group membership is determined according a predefined threshold based on traffic similarity. By using this framework, a group can contain VMs that are not exactly similar; the group will contain the common ACL and each VM will be configured individually to fulfill the remaining requirements.

1.2 Security-aware Resource Allocation in Clouds

In current VM provisioning processes, several factors (such as CPU, memory, storage, bandwidth) influence the final location of VMs. Unfortunately, security factors are not among them. Instead, security requirements are considered after resources are allocated (post-allocation). We argue that considering security requirement at the time of resource allocation can reduce the vulnerabilities that result from allocating VMs with conflicting security requirements on the same physical host. Also, it can reduce the cost associated with the use of security middleboxes such as IPSec and IDS devices. The cost reduction can be from minimizing VM migration due to security conflicts, or from reducing the communication costs to reach security middleboxes.

Our formal method-based approach enhances security in clouds by providing a security-aware resource allocation algorithm that takes security requirements in consideration along with network, performance, and space requirements [6]. The user provides the security and risk requirements for the IaaS VMs. Then, the cloud provider uses these requirements and our approach to allocate resources. The pre-allocation mechanism involves the user in the decision process and increases the awareness of user's security requirements.

1.3 Virtual Machine Migration Planning

The dynamic nature of clouds requires moving VMs from one host to another host to utilize resources efficiently. In some cases, the migration process may include

intermediate steps to migrate VMs to the final target host. During these steps, safety conditions – including risk, performance, and network requirements – must be preserved. Current cloud providers do not provide this guarantee. Therefore, it is required to find a VM migration sequence in which the safety requirements are not violated.

We offer a formal method-based approach to finding a safe sequence to migrate VMs to their final host [7]. The resulting VM migration plan ensures that security requirements along with performance and network requirements are preserved during the entire life cycle of a VM, including the times of migration. Moreover, finding a safe sequence to migrate VMs will also help in updating cloud configurations accordingly using a consistent mechanism. The correct and automated updates of configurations due to VM migration will reduce misconfigurations errors.

1.4 Virtual Machine Migration Verification

A successful VM migration process requires correctly updating cloud network middleboxes such as firewalls and switches to keep VMs work properly after migration. The migrated VMs need to maintain all previous connections with other VMs. Therefore, it is important to update network middleboxes at both locations, the old location before migration and the new location after migration. Updating and reconfiguring network middleboxes entails blocking all traffic to the migrated VMs at their old location and allowing the traffic – that was previously allowed at the old location – at the new destination. Migrating a VM to a new location may jeopardize its security policy due to misconfigurations that result from updating network middleboxes. For instance, unwanted traffic is allowed to reach the VM at the new location.

One approach to ensure proper reconfiguration is to execute automatically a comprehensive regeneration of all middlebox policies using VM reachability requirements and associated security policies. An alternative, more lightweight approach is to introduce incremental network reconfigurations to the existing network configuration.

This approach takes advantage of the previous network configuration - in particular, those portions that are unaffected by the migration - and builds upon it.

We offer a formal method-based approach to verifying the preservation of cloud network configuration after VM migration. The cloud network configuration is preserved if all migrated VMs maintain an equivalent network connectivity/policy after migration. Also, we offer an automated formal method-based framework for incremental reconfiguration of firewall policies to be executed as part of the VM migration process. The framework translates cloud network access policies into a set of firewall rules considering the physical topology of the cloud, the VMs to be migrated, and the VM migration plans.

1.5 Software Defined Networks Misconfigurations

Managing traditional datacenters (multi-tier architectures) is a challenging task. It requires a global view of the entire network to provide proper management decisions. The SDN paradigm provides a central and global view of the network through a centralized controller. For cloud infrastructures, SDNs provide crucial support for virtualization by making cloud management simpler. The SDN architecture is mainly formed from a centralized controller and distributed switches. The SDN controller manages all switches in the network and is responsible for updating and configuring the right policy for each switch in the network. Unfortunately, SDNs are susceptible to misconfigurations and errors like any other network type.

SDN functionality depends on its policy and configuration; any misconfiguration in the controller or switches can affect the behavior of the SDN. Network flow policies are translated into switch flowtables. Flowtables contain rules that perform actions. Misconfigurations in flowtable rules can lead to the execution of unintended actions. We propose formal method-based approach to detect misconfigurations in switch flowtables [8]. Misconfigurations can occur within the same flowtable (intra-policy), between different flowtables at the same switch (pipeline misconfigurations),

or between different flowtables at different switches (inter-policy). Our framework checks for all possible policy misconfigurations.

1.6 Methodology

Throughout this dissertation, we used two main formal model checking techniques to solve the outlined challenges. The techniques are Binary Decision Diagrams (BDDs) [9, 10] and Constraint Satisfaction Problems (CSPs) [11].

Access control lists and policies are modeled as Boolean formulas using BDDs. BDD modeling supports the use of algebraic operations such as intersection, union, and negation. BDD modeling is used to construct access control lists, i.e., reachability policies, for IaaS VMs. BDD modeling also provides a formal representation to solve misconfigurations in SDN policies. We use BDDs to model a user's incoming traffic policy and manage VM security groups. We also use BDDs to calculate the similarity between VM policies.

For VM migration verification problem, we model the cloud network access model as a set of access routes between pairs of VMs. An access route between a source VM and a destination VM defines a sequence of network devices in a single route between two VMs. This sequence of devices is used to compute the combined policy of all devices along this route. Each device policy is encoded as a BDD. The set of all accessible routes between two VMs is represented a BDD.

In SDNs, BDDs are used to: encode OpenFlow configuration considering the priority-based matching semantic, various actions, and the existence of multiple controllers and multiple users; model the global behavior of the OpenFlow network based on FlowTables over single or multiple federated infrastructures in a single state machine; and provide a generic property-based verification interface using BDD-based symbolic model checking and temporal logic.

We formalize security-aware resource allocation problem and virtual machine migration problem as CSPs that factor in the cloud provider's constraints, i.e., the

customer security and resource requirements, and the customer's risk metrics. The CSP problem formulation can be solved using a SMT solver (e.g., Yices [12]). Our solution for security-aware resource allocation [6] can be used to guide the grouping of VMs into security groups and the placement of virtual machines in a manner that satisfies a customer's security requirements, improves manageability, and reduces risk. Our solution [7] can also be used to migrate the virtual machines in the right order to preserve security constraints. Furthermore, it can also be used to update cloud network configurations that are affected by the migration process.

Each problem, resource allocation problem and VM migration planning problem, is encoded into a set of constraints. A SMT solver is then used then to find a solution that satisfies the constraints.

1.7 Contributions

Within this dissertation, we provide a formal method-based approach to provide better security and manageability of IaaS clouds environments.

This dissertation reports the following contributions.

- A formal method-based framework to synthesize access control lists for IaaS virtual machines.
- A formal method-based security-aware resource allocation methodology.
- A formal method-based framework for virtual machine migration planning.
- A formal method-based automated framework to verify configuration consistency before and after virtual machine migration.
- A formal method-based automated framework for post-migration reconfiguration.
- A formal method-based approach to modeling SDN OpenFlow configurations.

- A formal method-based verification interface for SDNs using BDD-based symbolic model checking and temporal logic.
- A formal method-based framework for detecting policy misconfigurations in SDNs.
- A unified approach to combine all previous frameworks under a common methodology.

1.8 Document Outline

This dissertation is organized as follows: Chapter 2 discusses the related work; Chapter 3 provides a background about the basic concepts discussed in this work; Chapter 4 introduces the proposed approach in detail; Chapter 5 illustrates the evaluation mechanism, the implementation, and the results; and Chapter 6 concludes the document with a summary of the proposed work and the potential work that could be pursued in the future.

CHAPTER 2: RELATED WORK

This chapter discusses related work associated with the primary problem areas of focus: managing access control policies for VMs, security-aware resource allocation in clouds; virtual machine migration planning; and SDN misconfiguration.

2.1 Managing Access Control Policies for Virtual Machines and Security-aware Resource Allocation in Clouds

Many recent studies – e.g., [13], [14], [15], and [16] – expose security risks associated with cloud environments. Somorovsky et al. [14] provide a security study in which the control interfaces of Amazon and Eucalyptus (private cloud software) are analyzed. The outcomes from this study show that the control interfaces in Amazon and Eucalyptus can be compromised using signature wrapping and advanced XSS techniques. Wei et al. point to the new risk coming from sharing virtual machine images in a cloud's image repository [16]. They propose an image management system to address sharing risk. The proposed system uses an ACL to reduce the unauthorized access, filters to remove unwanted information from the image, and a tracking mechanism for the operations applied on the image. Sumter proposed in [15] a design that captures the information flow on the cloud. A security analysis of Amazon's EC2 service is presented in [13]. In this analysis, an automated tool is used to analyze the security of Amazon's public Amazon Machine Images (AMIs). The finding of this analysis is that AMIs may be vulnerable to security risks such as unauthorized access, malware infections, and loss of sensitive information.

In light of the above studies, there is a heightened need for cloud providers to consider customer security requirements at the time of customer onboarding and

resource provisioning rather than afterward. The security-aware resource allocation framework we describe is a step toward addressing that need.

Several resource allocation approaches that consider various aspects of this problem have been published. Meng et al. present an algorithm for efficient resource provisioning via VM multiplexing [17]. The approach uses statistical multiplexing to find workloads patterns for VMs; VMs with complementary workload patterns are allocated together. Alicherry et al. present an approximation algorithm for VM placement in distributed clouds [18] and for allocation of resources in a single datacenter. Lee et al. [19] present a "what-if" methodology to allocate VMs. In their architecture, a prediction engine is used to estimate the performance, while genetic algorithms are used to find the optimal allocation. Goudarzi et al. present an SLA-based algorithm for resource allocation [20]. The algorithm uses forced-search to provide an upper bound limit to maximize the profit. Calcavecchia et al. introduce a backward speculative technique for VM placement in [21]. This technique projects the past demand of already allocated VMs to the host machine for future allocation decisions. Wei et al. [22] present a game theoretic approach, in which a constrained cost-time optimization algorithm for scheduling dependent subtasks considering the communications between these subtasks is presented. Srikantaiah et al. consider energy in their resource allocation algorithm [23], and present a heuristic approach for consolidating VMs so as to minimize the total energy consumption of the system. Other work considers minimizing energy while maximizing the profit [24].

While the above works consider a multitude of factors associated with ACL configuration and resource provisioning such as efficiency, performance, topology, cost, and energy during the resource allocation process, none combines ACL generation and security-aware resource provisioning within a single, effective methodology.

2.2 Virtual Machine Migration Planning

There are several efforts to address the problem of VM migration. The area of VM migration is driven by several factors such as: resource utilization, energy consumption, cooling and thermal concerns, network and application dependency, etc. Our approach utilizes the existing work that has been done in the area and complements it by providing a method for secure VM migration planning.

Piao et al. present a network-aware approach for placing and migrating VMs considering the network condition between physical hosts [25]. In this approach, VMs are placed and migrated to obtain shorter data access time. Shrivastava et al. propose an approach to migrate VMs that have application dependencies between them [26]. They present a greedy algorithm to place VMs on hosts considering the application dependencies between VMs and the cloud's physical network topology. Wood et al. [27] propose an approach that uses black-box and grey-box like memory monitoring, CPU utilization, and network utilization to detect hotspot hosts and migrate VMs to more suitable hosts. Ma et al. present a memory-encoding approach to decrease the total transferred data, migration time, and migration downtime by identifying the useful memory pages to be transferred [28]. Al Shayeji et al. present an algorithm to save energy in datacenters by migrating VMs off minimally utilized hosts [29]. Mishra et al. discuss the components of VM migration — when to migrate, which VM to migrate, and where to migrate — and different heuristics to apply VM techniques [30]. Zhang et al. outline the problem of VM migration in over-committed clouds [31]. They introduce an algorithm to minimize the number of VM migrations, to balance VM resource utilization, and to reduce the risk of overload in the cloud. Voorsluys et al. present a performance evaluation on the effects of live migration of virtual machines on the performance of applications running on "Xen" VMs [32].

None of these works considers the preservation of security requirements during VM migration planning.

The approach most similar to our approach is one by Ghorbani et al. [33]. They address the problem of determining the order of VM migrations. A heuristic approach is used to assign a migration score for each VM in the migration set. The score for VM_x reflects the number of migrations that would be feasible after the migration of VM_x. The approach considers the bandwidth limit as a migration constraint. Compared to our approach, we consider several constraints that include: capacity, dependency, security/risk, and VM workload characteristics.

Li et al. study the live migration features in load balancing scenario [34]. In their experiments, they show the effect of VM workload characteristics on the migration time and the migration downtime for different scenarios. We leverage the observations and remarks presented in their work to assign a priority score for each VM based on its workload.

2.3 Virtual Machine Migration Verification

The research in policy misconfigurations and anomaly detection is not new [4, 35, 36, 37, 38, 39]. Several works on configuration verification and equivalence checking have been done [40, 41, 42]. Gawanmeh et al. [40] use Event-B and invariant checking to verify the consistency of firewall verifications. Al-Shaer et al. [36] use Binary Decision Diagrams (BDDs) and symbolic model checker to verify network reachability and security properties. Gouda et al. [38] use firewall decision diagrams to verify the accept and discard properties of a given firewall policy.

Jarraya et al. [43] present cloud calculus, a framework that allows the expression of the cloud topology and the deployment and migration of VMs along with their security policies. Cloud calculus framework is used to verify the preservation of security constraints after migration. However, the framework does not support updating firewall policies after migration. Our approach allows generating the new post-migration configurations from the existing pre-migration configurations considering the VM migration updates.

The closest work to our work in post-migration verification is done by Jarraya et al. [44]. They proposed an automated approach to verify firewall configuration after VM migration. Their approach uses constraint satisfaction problems to verify the semantic equivalence of the defined properties. However, the approach does not address reconfiguring firewall policies after VM migration, it targets the verification of policy equivalence after VM migration.

Another work that addresses configuration verification in clouds is done by Eghtesadi et al. [45]. In this work, an automated framework is presented to verify the compliance of intrusion detection systems and IPSec polices after VM migration. The security policies are encoded as constraint satisfaction problems.

2.4 Software Defined Networks Misconfigurations

The concept of an OpenFlow switch as utilized in Software Defined Networks (SDNs) was introduced in [46] and used in different applications such as [47] and [48]. The work done on OpenFlow switches [46] does not address the problems of conflict analysis and model verification. Instead, it shows the basic architecture of OpenFlow model and how the architecture can be used to provide logically separated networks on the physical network. The work done in [47] introduces network virtualization in which a production network is sliced to multiple virtual networks that run multiple experiments (network instances) with their own forwarding decisions all at the same time.

Some work has been done on SDN configuration analysis. Flover [49] uses assertion sets and modulo theories to verify flow policies. As each flow rule request is verified against the non-bypass properties enforced in the network, Flover supports a batch mode to improve the controller response time. Veri-Flow [50] proposes to slice the OpenFlow network into equivalence classes to efficiently check for invariant property violations. VeriFlow models the network as a graph to detect loops in the routing tables, unavailable paths etc. Nice [51] uses symbolic execution to verify conformance

of OpenFlow applications. This solution enables the detection of when a network reaches an inconsistent network state. FlowGuard [52] presents a solution to check network flow path spaces to detect firewall policy violations when network states are updated.

Some work has been done on conflict analysis and model verification for devices other than SDN. For example, firewall modeling and conflict analysis was targeted by [35], [37], [36], and [39]. Also, there is considerable work on detecting misconfiguration in routing (e.g., [53], [54], [55], and [56]). Other works have been done on creating general model for network devices ([4], [57], and [58]).

Our work on SDN OpenFlow switch misconfiguration and model verification [8] uses BDD-based model checker and Computational Tree Logic (CTL) to write queries to describe properties that we are trying to verify. Our approach then uses BDDs to model a state machine that encodes the entire behavior of OpenFlow switches in a SDN. With our approach, we show that we are able to reduce the likelihood of SDN misconfigurations.

CHAPTER 3: FOUNDATION AND BACKGROUND

3.1 Cloud Computing

Cloud computing has become an essential solution for processing, managing, and storing large volumes of data. Cloud-based infrastructures are a main driver for the emergence of big data analytics. There are three main deployment models of cloud computing: private clouds, public clouds, and hybrid clouds. In private clouds, data and services are managed within the organization in a private network without outside exposure. In public clouds, data and services are located off-site and provided to the organization over the internet. In hybrid clouds, data and services are split across private and public environments.

There are three primary types of cloud computing solutions: Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS), and Infrastructure-as-a-Service (IaaS). SaaS Cloud providers host applications managed by third-party vendors. The applications are accessed via a web browser and do not require installation at the client side. Everything is managed by SaaS provider: applications, runtime, data, middleware, O/S, virtualization, servers, storage, and networking. A common example of a SaaS solution is the gmail web application.

PaaS cloud providers wrap computational resources in a platform. PaaS clients use these platforms to develop and customize applications. The PaaS model eliminates the need to buy the underlying layers of hardware and software. With PaaS, the cloud provider still manages the runtime, middleware, O/S, virtualization, servers, storage, and networking, but clients manage applications and data. GoogleApp Engine and Microsoft Azure are examples of PaaS solutions.

IaaS Cloud providers offer computing resources, storage, and networks for users

to setup their own virtual network of virtual machines, applications and data. IaaS cloud providers still manage virtualization, servers, hard drives, storage, and networking. Compared to SaaS and PaaS, IaaS clients are responsible for managing more: applications, data, runtime, middleware, and O/S. Consequently, there is a greater risk for security misconfigurations. Amazon EC2 as an example of an IaaS solution.

3.2 Binary Decision Diagrams (BDDs)

A binary decision diagram is a tree-like structure that represents a boolean function. Each boolean function is comprised of a set of decision variables. A BDD is modeled as a directed acyclic graph (DAG) that has a root node, one or more non-terminal nodes, and two terminal nodes representing the constants zero and one. Non-terminal nodes represent decision variables. Each non-terminal has two outgoing edges: a high edge and a low edge. The high edge represents a true assignment of the decision variable and the low edge represents a false assignment of the same decision variable [9].

Figure 3.1 shows an example BDD of the Boolean function $B = (x \wedge y \wedge \neg z) \vee (x \wedge \neg y \wedge z)$. The BDD in Figure 3.1 has three decision variables in the following order: x, y, and z, respectively. The solid line represents a true assignment (high edge) while the dashed line represents a false assignment (low edge).

There are many variations of BDDs depending on variable ordering. In this work, we use reduced ordered binary decision diagrams (ROBDDs) [9]. A ROBDD is a canonical representation of a boolean function. This means that there is only one ROBDD representation for a given boolean function. The sequence of decision variables along any path from the root to a terminal node in a ROBDD is guaranteed to follow a specific order. The BDD shown in Figure 3.1 is a ROBDD with the following order: x, y, z.

The BDD modeling supports algebraic operations such as intersection and negation on boolean functions.

$$B = (x \wedge y \wedge \neg z) \vee (x \wedge \neg y \wedge z)$$

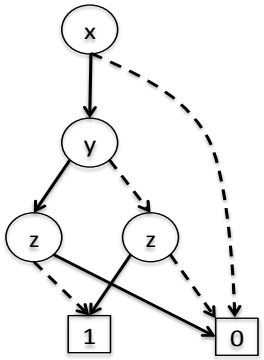

Figure 3.1: A BDD example

3.3 Software Defined Networks (SDNs)

SDNs consist of three layers: an application layer, a control layer, and a data plane layer [59]. Figure 3.2 shows the SDN architecture model. The Application layer consists of end-users applications. This layer allows users to program a SDN controller in the control layer through APIs. The SDN controller controls the forwarding behavior of the SDN network. The data plane layer includes the network devices, e.g., OpenFlow enabled switches and infrastructure that provide packet switching.

3.3.1 OpenFlow Switches

OpenFlow switches are used to partition a physical network to multiple logical networks (i.e., SDNs) [46]. OpenFlow switch clients (users) operate IaaS without conflict. Deploying OpenFlow switches on a physical network makes the network programmable. This gives the SDN provider the ability to control the traffic in a SDN. To prepare a physical network to operate multiple SDNs, cloud providers need to partition the traffic. Clients have the right to control only their own IaaS traffic.

To that end, each client (i.e., IaaS instance) is given a slice [47] on the network based on its needs. Operations on a IaaS instance should give a client the perception of operating on a separate physical network. Also, different IaaS instances should not

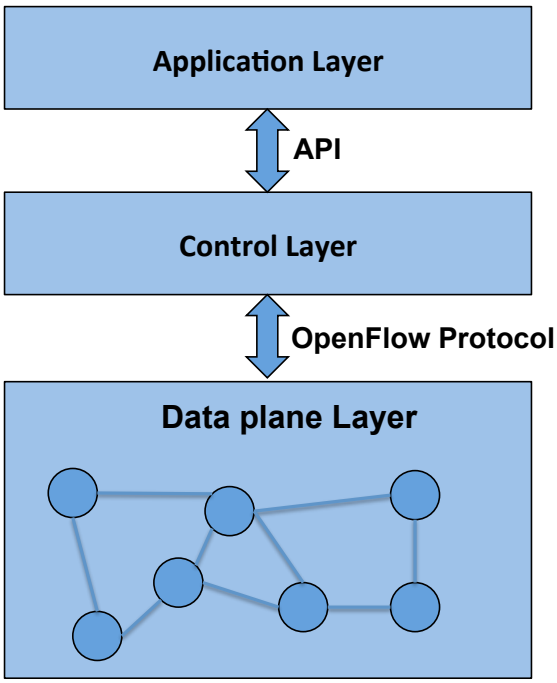

Figure 3.2: The SDN architecture model

overlap each other. Thus, one of the security challenges is to make sure that different IaaS instances do not overlap.

Conceptually, each SDN has four components: a controller, a secure channel, OpenFlow switches, and the OpenFlow protocol [46]. Figure 3.3 shows the main components of an OpenFlow switch. The OpenFlow protocol operates over a secure channel as an interface between controller and the OpenFlow switch [46]. As shown in Figure 3.3, the controller is used by the client (user) to update the FlowTable in an OpenFlow switch by adding and removing flow entries. Flow entries are actions used by a client to control SDN traffic. A FlowTable contains rules comprised of actions that work as commands on SDNs to impact SDN traffic. Potential actions include: forwarding, drop, encapsulate, encrypt, limit, and classify/enqueue for QoS [46]. It is possible for clients to add additional actions to a FlowTable via the OpenFlow protocol.

A FlowTable rule has three parts: header, action, and statistics. The header part is used to define a specific flow; the action part is used as a command for processing

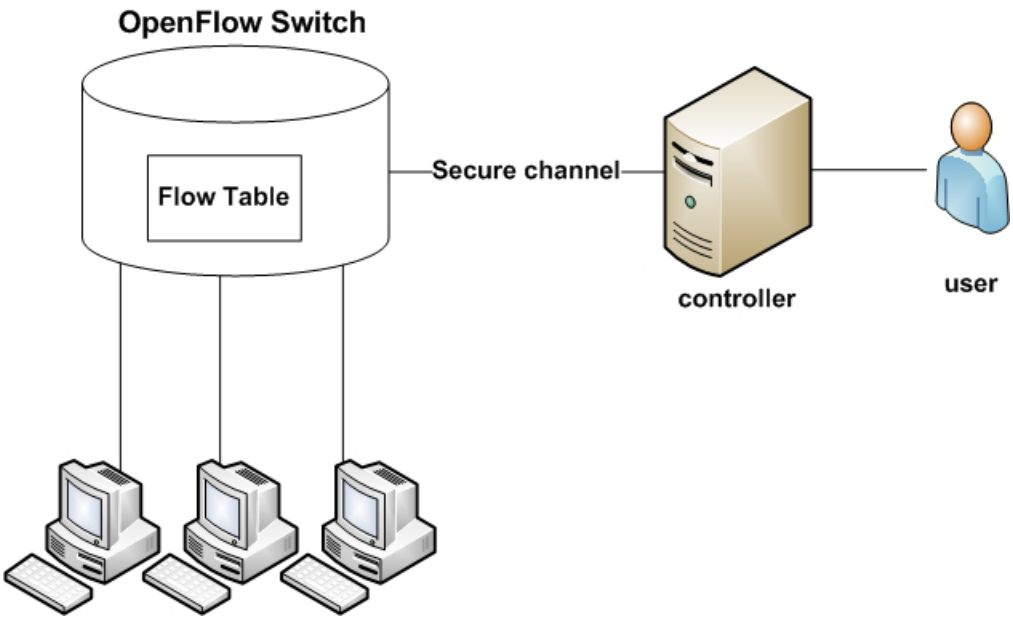

Figure 3.3: The basic components of an OpenFlow switch

purpose; and the statistics part is used for QoS operations. When a packet comes to OpenFlow switch, it is matched against the FlowTable rules. An associated action is triggered if the packet matches a rule header. If an action is executed, then the associated statistics are updated [46]. If the packet does not match a FlowTable rule, it is forwarded to the controller for more processing. If a packet matches multiple rules, the rule with the highest priority is triggered.

The OpenFlow protocol supports having multiple FlowTables within a single switch, which is referred to as a *pipeline*. The OpenFlow pipeline is composed of multiple "stops", or FlowTables, where various tasks are performed. These stops may perform multiple actions on a single packet.

In the pipeline, FlowTables at each switch are numbered starting from 0. The packet is matched against rules in table 0 first, then forwarded to next table in increasing order. The network packet is forwarded from one table to another table if its table ID is larger than current table ID, i.e., there is no backward processing. This forwarding is achieved using the "goto table" command. Having multiple "goto table"

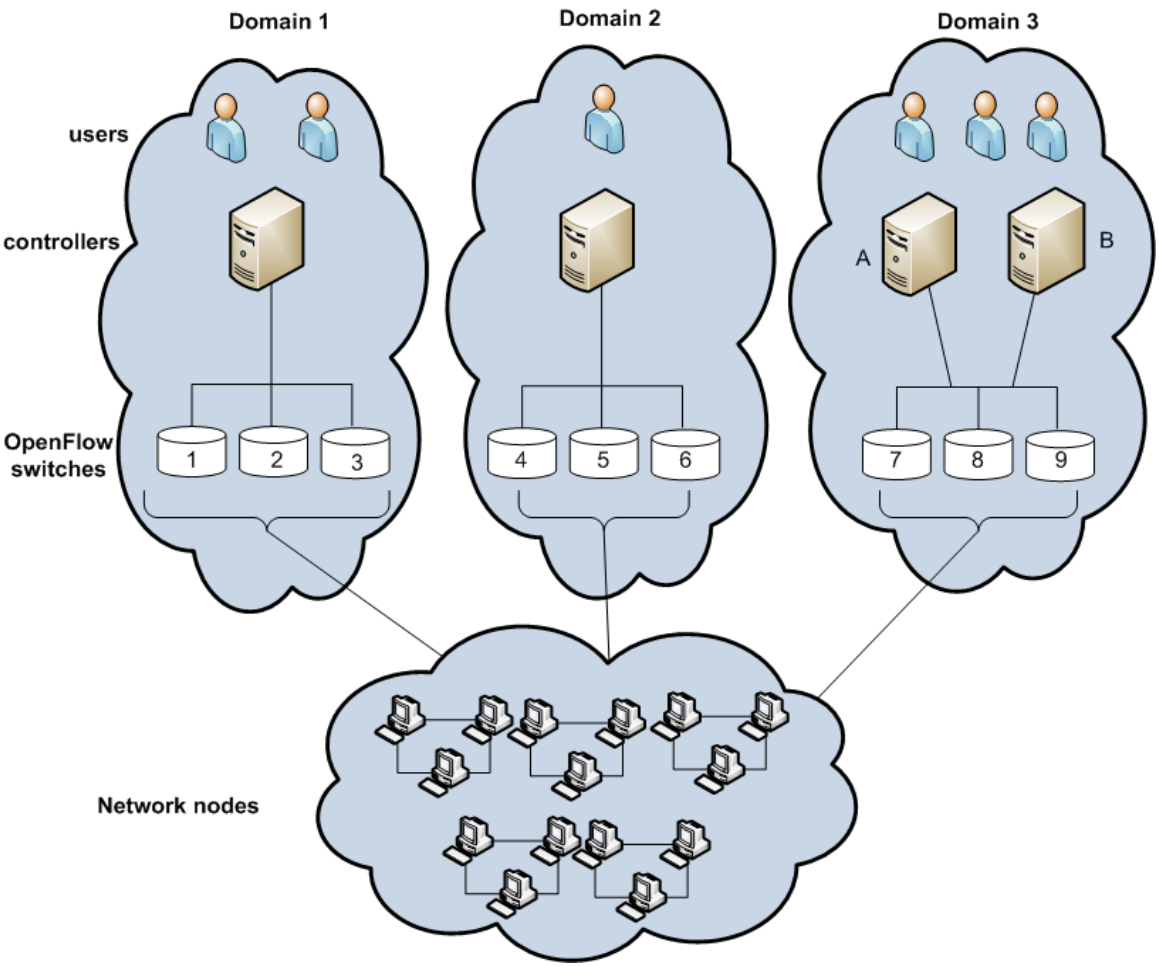

Figure 3.4: OpenFlow protocol allows multi-users and multi-controller management in IaaS cloud environments.

commands in different FlowTables within a single switch creates several pipelines and makes detecting policy misconfiguration a complex task.

In the face of multiple clients, the challenge of properly configuring SDNs is further magnified. The OpenFlow protocol allows more than one controller to control the same OpenFlow switch. It also allows one controller to manage multiple OpenFlow switches (see Figure 3.4). As a result, misconfigurations may appear in the FlowTables. The next chapter describes a common FlowTable misconfiguration (see §4.5).

CHAPTER 4: APPROACH AND METHODOLOGY

The primary goal of this work is to provide a unified set of approaches and methodologies that make IaaS cloud environments more secure and make cloud configurations management much simpler. To achieve this goal, we propose a formal method-based approach to provide better solutions to the following problems: 1) managing and creating access control lists for virtual machines running in IaaS clouds; 2) allocating resources for virtual machines considering security and risk requirements; 3) migrating virtual machines from one host to another host without violating security requirements; 4) verifying that cloud network configuration is preserved after migration and reconfiguring cloud networks efficiently to reflect the new state after VM migration; and, 5) detecting policy misconfigurations in Software Defined Networks.

A glossary of all variables used in the presentation of our approach is shown in Table 4.1.

4.1 Managing Access Control Policies for Virtual Machines

In order to manage ACLs effectively and to perform security-aware provisioning of required resources to VMs, we need to organize VMs into security groups based on VM similarity.

4.1.1 Problem Statement

IaaS Cloud clients utilize cloud environments to deploy virtual networks. An essential part of any IaaS deployment is proper specification and management of reachability requirements between VMs. Reachability requirements are designed to control how VMs communicate with each other. IaaS cloud clients are responsible for translating reachability requirements into ACLs. This activity requires clients to produce

Table 4.1: Glossary of all variables used in the model

Variable	Type	Description
VM	Set	The set of running VMs
PM	Set	The set of physical machines (PMs)
H	Set	The set of "'hotspot" (PMs) that is scheduled to be switched off
M	Set	The set of VMs to be migrated
R	Set	The set of risk scores for all VMs
C	Set	The set of critical VMs
D	Matrix	The dependency matrix between VMs
n	Integer	Number of virtual machines
m	Integer	Number of physical machines
$a_{i,j}^{x}$	Boolean	Decision variable to place VM_i on PM_j at the x^{th} migration step
$a_{i,j}$	Boolean	Decision variable to allocate VM_i on PM_j
c_i^{cpu}, c_i^{mem}	Real	*cpu/memory* capacity for VM_i, respectively
C_j^{cpu}, C_j^{mem}	Real	*cpu/memory* capacity for PM_j, respectively
s_j	Boolean	A variable to indicate the status of PM_j
o_i	Integer	The priority score of VM_i
π_x	Set	The placement status for all VMs at the x^{th} migration step
$\rho(\pi_x)$	Formula	The safety condition for the placement status π_x
η_i	Boolean	A variable to flag critical VMs
$\delta_{i,j}^{k}$	Boolean	A variable to indicate that VM_i is migrated to PM_j at the $(k+1)^{th}$ migration step
r_i	Enumeration	The risk score for the VM_i
$l_{v,w}$	Integer	The physical distance between PM_v and PM_w
T_D	Integer	A threshold to set the max physical distance between two dependent VMs
g	Integer	Number of security groups
$s_{i,k}$	Boolean	Decision variable to allocate VM_i on security group k (SG_k)
$d_{i,j}$	Integer	The distance between VM_i and VM_j
$D_{x,y}$	Integer	Number of links between PM_x and PM_y
$b_{i,j}$	Boolean	Decision variable to indicate whether VM_j is reachable from VM_i or not
B_i	BDD	BDD representation for VM_i
$\mathcal{T}$	Integer	The maximum number of iptable rules in a VM
t_1, t_2	Integer	Thresholds to set min/max values for risk categories
$P_{i,j}$	Boolean	Decision variable to indicate the traffic between VM_i and VM_j is to be inspected
$Q_{i,j}$	Boolean	Decision variable to indicate that VM_i and VM_j are not be hosted on the same PM
E	Integer	QoS threshold to limit distance between PMs
r_i	Real	Residual risk for VM_i
R_k	Real	Residual risk for SG_k
G_k	Integer	Vulnerability score for SG_k
v_i	Integer	Vulnerability score for VM_i
$\widehat{I_k}$	Integer	Impact value for SG_k
I_j	Integer	Impact value for VM_j
l_i	Integer	The cost for VM_i to access the closest middlebox
L	Integer	The maximum cost for inspecting traffic between two VMs

and manage a separate ACL for each VM - a process that can be both tedious and complex. This process, however, can be simplified by grouping VMs that share similar reachability requirements. Unfortunately, even with shared ACLs this process is subject to misconfigurations.

To combat this problem, we present a semi-automated, formal methods-based framework for creating security groups and managing ACLs for VM security groups. Under our framework, VMs are group based on their incoming traffic similarity. The framework allows VMs to be grouped whose incoming traffic requirements are not identical. A threshold is defined to limit the degree of difference among reachability requirements that is permitted for each VM security group. After grouping VMs, the framework generates: (i) an ACL for each security group; and, (ii) a set of iptable rules for each VM to manage the variance among reachability requirements within the VM security group.

The proposed framework reduces the likelihood of conflicting ACLs and/or misconfigured ACLs. Furthermore, each VM security group serves as logical zone designed to reduce residual risk for VMs.

4.1.2 Framework Overview

The security group membership assignment framework is shown in Figure 4.1. The framework includes three stages. In the first stage, the client specifies the reachability requirements for each VM. The reachability requirements define how VMs can communicate with each other. The reachability requirements are encoded as a reachability matrix. Table 4.2 shows an example reachability matrix. In this table, 0 means that two VMs are not reachable, 1 means that two VMs are reachable using any port number, and $\{x\}$ means that two VMs are reachable using port number (x). The second half of stage one involves the automatic construction of a distance matrix. A distance matrix measures the similarity between VMs according to incoming network traffic. More details about the distance matrix is discussed in §4.1.3.

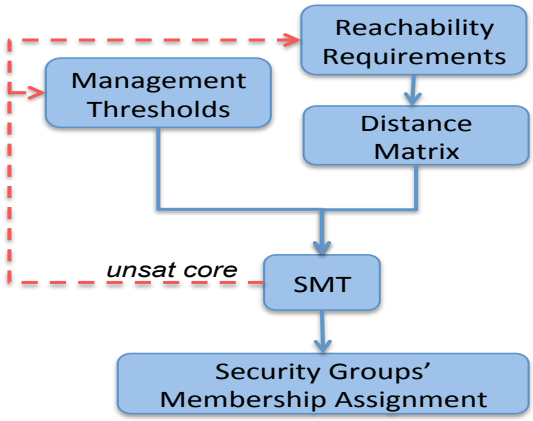

Figure 4.1: Security groups membership assignment framework

Table 4.2: An example of reachability requirement matrix.

		Distination			
		VM_1	VM_2	...	VM_n
Source	VM_1	0	1	...	{80}
	VM_2	{22}	0	...	0
	⋮	⋮	⋮	⋱	⋮
	VM_n	{22, 80}	1	...	0

The second stage of the security group membership assignment framework is the encoding of the security group model that will be used to assign VM security group membership. The security group model is encoded as constraints. An example security model constraint might be the enforcement that all VMs to be a member of at least one security group. Management thresholds are also defined as security model constraints. An example of a management threshold constraint is $\mathcal{T}$, the maximum number of iptable rules for a single VM. More detail about the security group model and encoded constraints is provided in §4.1.4.

In the final stage of the security group membership assignment framework, the distance matrix, security group model, and the management thresholds constraints are inputted into the SMT solver to generate automatically security group membership

assignments for each VM. If the SMT solver determines that all the constraints cannot be simultaneously satisfied (i.e., *unsat core*), the reachability requirements and/or management thresholds must be relaxed.

4.1.3 Distance Matrix

The framework utilizes BDDs to encode incoming traffic policies for each VM. BDD modeling gives us the ability to employ useful algebraic operations such as intersection and negation [9, 10]. These operations are the basis for the distance metric. The distance metric is calculated based on aggregate incoming traffic to all VMs. The computed metric captures the main function of security groups in Amazon EC2 [5]. However, all members in an Amazon EC2 security group permit the same incoming traffic. Therefore, it is possible within an Amazon EC2 security group to find a VM accepting more traffic than what is supposed to accept. In this case, we call the unwanted traffic *spurious* traffic.

There are two limitations in Amazon EC2 security groups model that permits spurious traffic [5]. First, the rule structure used in writing security group ACLs has three fields to control traffic: protocol, source IP address, and destination port number. This can negatively impact VM security because of a lack of fine grain control over destination addresses. Second, all rules are accept rules, there are no deny rules except the default deny rule. Thus, no exceptions can be made using Amazon EC2 security group ACLs.

Furthermore, spurious traffic jeopardizes VM security because of the additional accepted traffic that may conflict with the security policy for a given VM. One way to solve this problem is to use iptable rules to deny the spurious traffic. Iptable rules operate as firewall rules for a given VM. Each flow that correlates to spurious traffic needs an iptable rule to deny it. The presence of iptable rules, however, degrades the performance of the IaaS proportionally to the number of rules.

The challenge here is to keep the number of iptable rules for a given VM within a

specified limit (threshold $\mathcal{T}$) while properly modeling the permitted incoming traffic. This can be done by grouping VMs having similar incoming traffic.

Definition 1 *The distance between two virtual machines, $d_{i,j}$, is the size of the traffic accepted by VM_i and denied by VM_j.*

The incoming traffic at a single VM is modeled as a BDD. Security group rule structure is used to model acceptable incoming traffic for each VM. The BDD representation for a single security group filtering rule, f_z, encodes *three* fields: protocol, source IP address, and destination port number. Formally:

$$f_z \rightsquigarrow protocol \ \wedge \ source\_address \ \wedge \ dest\_port\_number$$

$$B_x = \bigvee_{z \in filtering \ rule} f_z$$

where B_x is the BDD representation for all incoming traffic rules for VM_x. Rule ordering in B_x has no effect because all rules have the same action; accept the incoming traffic.

The distance, then, between two VMs, VM_i and VM_j, is denoted as $d_{i,j}$. Definition 1 represents this formally as follows:

$$d_{i,j} = pathCount(B_i \wedge \neg B_j) \tag{4.1}$$

Where BDD operations ($\wedge$) and ($\neg$) represent intersection and negation operations, respectively.

Equation 4.1 finds the count of all traffic flows accepted by B_i and denied by B_j; *pathCount(x)* function finds the number of paths leading to the *true* leaf node in BDD B_x. Each path leading to the true node represents a satisfiable solution. The advantage of using the *pathCount(x)* function is that wildcard fields in a single path are counted once.

4.1.4 Synthesizing Security Groups

In an IaaS cloud environment, VMs often interact with one another to provide desired functionality. Constraints on this interaction are encoded as reachability requirements and represented in a reachability matrix. Table 4.2 shows an example of a reachability matrix. In this table, 0 means that two VMs are not reachable, 1 means that two VMs are reachable using any port number, and $\{x\}$ means that two VMs are reachable using port number (x).

The mechanism for enforcing reachability requirements is known as security groups. Among cloud providers, there are different mechanisms for the specification of security groups. Without loss of generality, we use the Amazon EC2 security group mechanism to illustrate our research contribution.

VM operation is affected by security group membership. A VM can belong to more than one security group [5]. Placing VMs into security groups without proper consideration of reachability and security requirements can jeopardize the security of all VMs. Therefore, grouping VMs into security groups is the first step toward securing an IaaS instance.

The distance matrix, D, is the basis of our method for synthesizing security groups and assigning VM membership. The distance metric is used to constrain security groups membership: *the pair-wise distance between all VMs in a security group should not exceed a specific threshold* $\mathcal{T}$. To synthesize security group rules, we use SMT solver to find VM membership based on the following constraints:

Membership Constraint. A VM can belong to multiple security groups. The following equations ensure that each VM will be mapped to at least one security group. The variable $s_{i,k}$ is a Boolean variable that indicates the membership of VM_i in security group SG_k.

$$\forall i \in VM, \forall k \in SG \quad s_{i,k} \in \{0,1\} \tag{4.2}$$

$$\forall i \in VM \quad \sum_{k=1}^{g} s_{i,k} \geq 1 \tag{4.3}$$

Constraint 4.3 ensures that each VM will have membership in at least one security group. We can restrict a VM to be member of one security group by changing the ($\geq$) operator in the constraint to the ($=$) operator.

Distance Constraint. Two VMs cannot be placed in the same security group if the distance measure between them is larger than the specified threshold $\mathcal{T}$. The threshold $\mathcal{T}$ is used to set the maximum number of iptable rules that can be used to block traffic for a single VM.

$$\begin{aligned} \forall i, j \in VM \quad & (d_{i,j} > \mathcal{T} \vee d_{j,i} > \mathcal{T}) \rightarrow \\ & \exists k, l \in SG\,(s_{i,k} \wedge s_{j,l} \wedge k \neq l) \end{aligned} \tag{4.4}$$

Access Control Policy Model Constraint. Allowing two different security groups to communicate with each other means allowing any VM of one security group to communicate with any VM of the other security group. This constraint is enforced by the format of security group rules. Amazon EC2 [60], for example, includes the security group name in the source address field to open a port for communication between security groups.

$$\begin{aligned} \forall i, j \in VM \quad & \Big(b_{i,j} \neq 0 \wedge s_{i,k} \wedge s_{j,l} \wedge k \neq l\Big) \rightarrow \\ & \exists x, y \in VM \Big(b_{x,y} \neq 0 \wedge s_{x,k} \wedge s_{y,l}\Big) \end{aligned} \tag{4.5}$$

The reachability between two VMs, VM_i and VM_j, is represented by the variable $b_{i,j}$.

4.1.5 Residual Risk for Security Groups

Managing the risk associated with security group interaction is a major factor in implementing a *defense-in-depth* strategy. A defense-in-depth strategy suggests using

multiple security techniques to mitigate the risk of compromised machines [61]. A common first layer in a defense-in-depth strategy is to create logical zones for the deployment of VMs. Security groups can define these logical zones that can be used as this first layer in a defense-in-depth strategy. The second layer in a defense-in-depth strategy is to deploy security devices such as IDSs and IPSec gateways to control interaction by VMs in different zones.

Thus, to synthesize security groups in a secure manner, we need to consider the residual risk introduced by the allocation of VMs to security groups. Residual risk can be estimated by considering two measures: 1) VM vulnerability score and 2) VM impact score.

The vulnerability score of a security group, therefore, is determined by the vulnerability of all VMs belong to this group. Without loss of generality, we consider the highest VM vulnerability score to be assigned for the vulnerability for the whole group. Formally:

$$G_k = max(v_j), \quad \{j | \, VM_j \in SG_k\} \tag{4.6}$$

where v_j, $0 \leq v_j < 1$, is the vulnerability score for VM_j. The vulnerability score for a single VM is determined based on the widely accepted Common Vulnerability Scoring System (CVSS) [62].

The impact score of a security group measures the potential damage for all VMs belonging to a security group. Formally,

$$\widehat{I_k} = \sum_{VM_j \in SG_k} I_j \tag{4.7}$$

where $\widehat{I_k}$ is the total impact for SG_k and I_j is the impact for VM_j. The impact score is a function of damage cost (data sensitivity) and the attack propagation from one VM to another VM. Our method for impact calculation is similar to Ahmed et al. [63]. The impact score for a VM is computed based on sensitivity of the VM data

and the cost of damage to the business resulting from the VM insecurity.

Using VM vulnerability and impacts scores, the following equation calculates the residual risk for a security group, SG_s, resulting from its exposure to other security groups by combining these two VM risk measures.

$$R_s = \frac{\sum_{i=1}^{g} y_{is} * G_i}{\sum_{i=1}^{g} G_i} * \widehat{I}_s \tag{4.8}$$

where R_s is the residual risk for SG_s, y_{is} is a Boolean variable indicating the reachability between two security groups SG_s and SG_i, G_i is the vulnerability score for SG_i, and $\widehat{I}_s$ is the impact score of SG_s.

4.1.6 Residual Risk for Individual Virtual Machines

Just as residual risk can be calculated for each security group, residual risk can also be calculated for each VM. The residual risk for a VM is determined by its security group membership. In our approach, the residual risk for a single VM inherits the residual risk value from the security groups of which it is a member. In other words, all VMs belong to the same security group inherit the same residual risk value.

Definition 2 *The residual risk for an individual VM is equal to the maximum of the residual risks of the security groups to which it belongs.*

Formally:

$$r_i = max(R_k), \quad i \in VM, k \in SG, s_{i,k} = 1 \tag{4.9}$$

Specifying a risk value for each VM opens the door for risk-aware resource provisioning. In addition to allocating VMs to physical machines based on capacity and performance metrics, residual risk values can be used to determine which VMs to allocate to the same host in order to minimize risk, thus adding another layer to a defense-in-depth strategy. For example, forcing a high risk VM and a high impact VM to be allocated to different hosts can add another layer of defense-in-depth.

4.2 Security-aware Resource Allocation in Clouds

Cloud providers may specify criteria such as availability, performance, cost, data security, and disaster recovery as part of their Service Level Agreements (SLAs). While the cloud provider may utilize a long list of technologies and products to ensure the security of the cloud infrastructure, the number of security control knobs made available to the cloud customer may be limited. For example, VPNs, VLANs, encryption of stored/transmitted data may be offered by the cloud provider to the cloud customer, whereas more advanced intrusion prevention, detection, and avoidance technologies may be entirely the customer's responsibility. Within an in-house data center, customers may be able to install firewalls to partition their network into dozens or even hundreds of security zones. However, such flexibility is not common on IaaS clouds today.

4.2.1 Problem Statement

Typical parameters specified by the IaaS cloud customer at the time of resource provisioning include the number of VMs and performance requirements such as CPU, memory, network bandwidth, storage capacity, etc. Based on the number of VMs and these requirements, cloud providers allocate VMs to available physical resources. Currently, however, cloud providers do not reveal how allocations are done. One can reasonably assume that the allocation is done in a way to maximize resource usage and minimize costs to the provider (e.g., through economies of scale). Security specifications at the time of resource request currently are limited to basic options such as anti-collocation constraints, i.e., the customer may specify that a virtual machine should not be collocated on the same physical server as another virtual machine (e.g., IBM SmartCloud Enterprise® [64]).

We consider the problem of *security-aware resource allocation* – i.e., resource allocation that factors in a customer's security requirements in additional to the regular

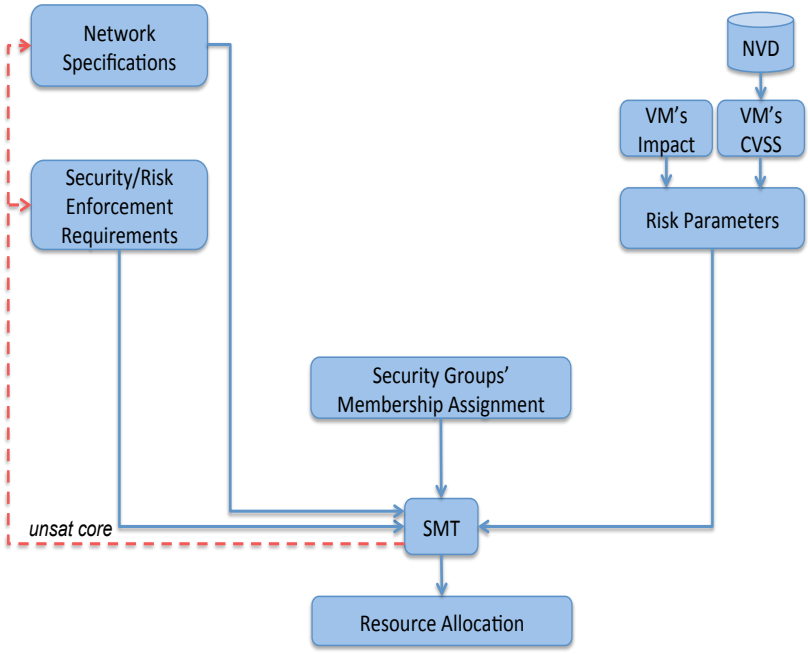

Figure 4.2: Resource allocation framework

resource specifications – with a focus on reachability and (network flow) access control for the IaaS VMs.

Fortunately, the problems of managing VM ACLs and performing security-aware resource allocation are closely related. It turns out that the automatically generated security groups that are used to reduce risk not only can positively impact ACL manageability, they also can support security-aware resource allocation. The allocation process, however, requires different types of inputs as shown in Figure 4.2. To support security-aware resource allocation, security enforcement constraints are defined to enable allocation decisions that will increase security levels and add extra security layers to a defense-in-depth strategy.

4.2.2 Framework Overveiw

The framework for security-aware resource allocation (Figure 4.2) works in three stages. The first stage collects input data and translates the data into constraints. Input data encompass: (i) cloud provider constraints; (ii) the customer resource re-

quirements; (iii) the customer security requirements; and (iv) the customer risk measures.

Cloud provider constraints include the number and size of physical and virtual resources, the cloud topology information (e.g., a three-tiered fat-tree topology), the security mechanism used for grouping VMs into logical security zones (e.g., EC2's security groups), and cloud management thresholds such as limits on the number of security groups (g), the number of rules per group, and the maximum number of iptable rules ($\mathcal{T}$) that can be written in a single VM.

Customer resource requirements include the number of VMs and performance requirements about CPU, memory, network bandwidth, storage capacity, etc.

Customer security requirements include VM reachability restrictions, security/risk enforcement specifications, and anti-collocation constraints. Security/risk enforcement specifications define when and how the security actions are enforced between the interacting VMs. Table 4.3 shows an example of a security/risk enforcement specification, which defines the type(s) of security mechanism(s) to be employed based on the risk level of two communicating VMs. In this example, *access*, *deny*, *inspect*, and *encrypt* mean allowing, denying, inspecting, and encrypting communications between VMs, respectively. *Anti-collocate* means provisioning VMs to different physical machines (PMs).

Table 4.3: An example of a security enforcement policy

		Risk (To)		
		High	*Medium*	**Low**
Risk (From)	*High*	inspect	encrypt	deny
	Medium	encrypt	access	anti-collocate
	Low	deny	anti-collocate	access

Customer risk metrics include VM vulnerability scores and impact scores. The vulnerability scores can be obtained from the National Vulnerability Database (NVD), which provides the Common Vulnerability Scoring System (CVSS) scores for almost

all known vulnerabilities [62]. The impact score [63] for a VM can be computed based on sensitivity of the VM data and the cost of damage to the business resulting from the VM insecurity. Once input data are collected, the data are translated into satisfiability constraints to complete stage one.

Stage two of the framework for security-aware resource allocation involves automatic VM assignment to security groups as previously discussed in §4.1.

The final stage of the framework for security-aware resource allocation is the allocation of resources to the requesting VMs. In this stage, the various input data such as security groups membership, network specifications, risk parameters, and security enforcement requirements are translated into satisfiability constraints and inputted into the SMT solver.

The problem is formulated as a satisfiability problem not as an optimization problem. Our objective is to find a assignment that satisfies both the customer requirements and the cloud provider requirements. The solution will not necessarily be the optimal one, but it is guaranteed to satisfy all input constraints if such a solution exists.

The proposed framework for security-aware resource allocation is both flexible and extensible. It does not require complete specification of all input data (i.e., (i) cloud provider constraints; (ii) the customer resource requirements; (iii) the customer security requirements; and (iv) the customer risk measures) in order to generate useful results. Moreover, additional user-defined constraints may be included to influence resource allocation in desirable ways. The additional constraints can be used to define extra security layers in a defense-in-depth strategy.

4.2.3 Resource Provisioning

Resource sharing and multi-tenancy are fundamental principles to the design of cloud infrastructures. In cloud environments, resources are designed be shared between multiple VMs - e.g., resources are linked to physical machines but utilized

by multiple virtual machines. When a VM requests a resource, the cloud provider satisfies the request by assigning a VM to a physical machine that is linked to the resource. This process is called provisioning. When provisioning resources, cloud providers must satisfy all resource requests while ensuring a proper isolation between the shared resources.

Within our framework for security-aware resource provisioning, we define the following constraints to preserve the integrity of the provisioning process.

Boolean Decision Variables. In a cloud environment, a decision variable is needed to control the assignment of virtual machines to physical machines (PMs). This decision variable is a Boolean variable defined as follows:

$$\forall i \in VM, \forall j \in PM \quad a_{i,j} \in \{0,1\} \tag{4.10}$$

The variable $a_{i,j}$ indicates if the VM_i is mapped to the PM_j.

Mutual Exclusion. In a cloud environment, a physical machine hosts zero or more virtual machines. A virtual machine, however, may be hosted by exactly one physical machine at any given time.

$$\sum_{j=1}^{m} a_{i,j} = 1 \tag{4.11}$$

Constraint 4.11 ensures that for every virtual machine VM_i, there is only one decision variable, $a_{i,j}$, set to *one*. The mutual exclusion constraint sets all other decision variables to *zero* for a given VM once a physical machine is assigned.

Capacity Limits. In a cloud environment, the capacity of VMs hosted in a single PM should not exceed the capacity limits of that PM. CPU, memory, disk space and bandwidth are the main specification dimensions to be used in resources provisioning. Without loss of generality, we consider only CPU and memory dimensions in the current framework. Constraint 4.12 represents the *CPU* constraint ensuring that capacity limitations will not be violated.

$$\forall j \in PM \quad \sum_{i=1}^{n} a_{i,j} * c_i^{cpu} < C_j^{cpu}, \quad i \in VM \tag{4.12}$$

where variables c_i^{cpu} and C_j^{cpu} represent CPU specs for VM_i and PM_j, respectively.

This constraint limits the total CPU capacity of VMs hosted on a single physical machine to the CPU capacity of the physical machine. Similar constraints can be defined for other resources such as memory, disk space, and bandwidth. The generality of the framework allows more than one resource dimension to be considered at a time.

While Constraints 4.10, 4.11, and 4.12 model VM resource requirements, the following constraints model VM security requirements. Both sets of constraints are used during security-aware resource provisioning.

Security requirements are defined to enhance the overall security for the IaaS using a defense-in-depth approach. Several security devices can be used to enhance the security: firewalls, IDSs, and IPSec gateways. Firewalls are used to control the traffic between VMs. Firewall requirements are modeled by the security groups and iptable rules. IDSs are used to inspect the traffic between VMs. IPSec gateways are used to encrypt/decrypt the communication between VMs.

In this discussion, we use the residual risk equations presented in §4.1.5 and §4.1.6 to add more layers to a defense-in-depth security strategy. Risk and impact values are organized into three categories: high, medium, and low. Thresholds, t_1 and t_2, are defined to set boundaries for each category. VMs that have risk values greater than t_2 are considered *high* risk VMs, VMs with risk values less than t_1 are considered *low* risk VMs. *Medium* risk VMs have a risk value between t_1 and t_2.

Security requirements are given by the user as shown in Table 4.3. Each cell in Table 4.3 becomes a security constraint within our model. We illustrate two such constraints here: inspection enforcement ($P_{i,j}$) and anti-collocation enforcement ($Q_{i,j}$).

Inspection Enforcement. The inspection enforcement ($P_{i,j}$) constraint specifies that traffic is to be inspected between two high risk VMs:

$$r_i > t_2 \wedge r_j > t_2 \rightarrow P_{i,j} \tag{4.13}$$

Formally, inspection enforcement is represented as follows:

$$\forall i, j \in VM \qquad P_{i,j} \in \{0, 1\} \tag{4.14}$$

$$\forall i, j \in VM \qquad P_{i,j} \rightarrow \exists x, y \in PM \, (a_{i,x} \wedge a_{j,y} \wedge x \neq y \wedge l_x + l_y < L) \tag{4.15}$$

In Constraint 4.15, l_x is the cost for PM_x to access the closest inspection middlebox. The inspection enforcement constraint requires that both of the communicating VMs be allocated on different hosts that can access an IDS middlebox with an acceptable cost L. Providing a threshold L for the inspection cost gives preference to those PMs close to IDS middleboxes to host VMs requiring traffic inspection. The cost L can be defined as the number of links needed to reach the IDS middlebox. We used a similar cost function to the one defined in [65].

This cost function increases the total cost as the distance between the hosting machine and the IDS middlebox increases. Preference is given to place VMs subject to this constraint on PMs that are closer to IDS middleboxes. Thus, for cloud providers, placement of IDS middleboxes within the cloud infrastructure is an important design consideration. Given the typical hierarchical structure where the lowest level contains the hosting PMs, placing IDS middleboxes high in this structure reduces the number of required middleboxes but increases the distance to the PMs. Conversely, placing the IDS middleboxes close to the PMs reduces the distances but increases the number of required middleboxes.

Anti-collocation Enforcement. Anti-collocation enforcement ($Q_{i,j}$) adds an extra layer of defense-in-depth. In this layer, an extra level of physical isolation is enforced by allocating VMs on different PMs within a QoS threshold.

Formally, anti-collocation enforcement is represented as follows:

$$\forall i, j \in VM \qquad Q_{i,j} \in \{0, 1\} \tag{4.16}$$

$$\forall i, j \in VM \qquad Q_{i,j} \rightarrow \exists x, y \in PM \, (a_{i,x} \wedge a_{j,y} \wedge x \neq y \wedge D_{x,y} < E) \tag{4.17}$$

In Constraint 4.17, $D_{x,y}$ represents the distance (number of links) between PM_x and PM_y, E is a threshold for QoS checking. Cloud providers can use risk-aware enforcement across multiple users to provide secure isolation.

4.3 Virtual Machine Migration Planning

VM migration is the process of changing VM assignment from one physical machine to another. VM migration is an elementary part in managing cloud environments. The VM migration planning problem considers the case in which there is a temporal relationship between VM migration steps. Several factors affect the temporal relationship and consequently the sequence in which VMs are migrated. One factor that influences the temporal relationship is physical network bandwidth limits. The optimal migration plan from an efficiency standpoint is to migrate all VMs simultaneously. However, bandwidth capacity typically limits the number of VMs that can be migrated at any one time.

A second factor that influences the temporal relationship of VM migration planning is collocation dependencies between VMs. For example, consider a situation in which an application VM and a database VM are running on two different PMs. The application VM has frequent data-intensive communication to the database VM; thus, it is desirable to allocate both VMs to PMs in close physical proximity to one another for performance reasons. During the VM migration process, the sequence of VM migrations may cause communication latencies if the collocation dependency is not enforced, which can lead to SLA violations. An example of such requirements is shown in §4.3.1.1.

Other factors that influence the temporal relationship of VM migration planning are related to security configurations and workload characteristics. For example, migrating one VM before another can result in a loss of communication between both VMs due to the security context of a VM that depends on its old location [66]. The workload characteristics of VMs affect the migration cost in terms of migration

downtime and migration transfer time [34]. As observed in the experiments done by [34], a memory-dirty workload VM or a web server workload VM will affect those VMs migrated before them; therefore, it is recommended to give higher priority for memory-dirty workload VMs and web server workload VMs to be migrated before other VMs to reduce the cost of migrations.

All of these examples demonstrate that VM migration planning is both complex and, at the same time, vital to the proper operation of IaaS instances. It is important to avoid cases in which the migration process incurs extra cost or violates resource, performance, or security requirements.

4.3.1 Problem Statement

Virtual machine migration planning is the problem moving VMs among PMs while satisfying temporal relationships associated with bandwidth, collocation, security, and workload constraints. In the following, we formalize the problem and provide an illustrative example.

Let PM, $|PM| = m$, represent the set of available physical machines (PMs). PMs are specified by their resources, e.g., CPU, memory, disk space, etc., capacities. Let VM, $|VM| = n$, represent the set of running virtual machines. VMs are similarly specified by their resources requirements such as CPU, memory, disk space, etc. Without loss of generality, we consider only CPU and memory resources dimensions in this presentation.

Let M, $M \subset VM$, represent the set of virtual machines to be migrated. Let H, $H \subset PM$, represent the set of hotspot physical machines that are scheduled to be switched off. Let D represent the dependency matrix for the running VMs. In D, $D_{x,y} = 1$ means that both VM_x and VM_y have communication dependency. Let C, $C \subset VM$, represent the set of critical VMs; these are VMs that cannot be migrated or reallocated after their first allocation assignment.

Let R represent the risk score for all VMs. R_i is the risk score for VM_i. Risk scores

for VMs depend on several factors such as: reachability and connectivity between VMs, the vulnerability of VMs, and the impact value (cost of damage) for VMs. More details about calculating VM risk scores can be found in §4.1.6 and [6]. In this work, the risk constraints require a VM to be placed on a physical machine (PM) that hosts VMs with a similar risk score. Risk constraints are also required to be enforced during intermediate migration steps.

Let π_x represent the placement status for all virtual machines at time x, π_0 represents the initial placement, and π_τ represents the target placement. Let σ represent the transition from one placement status to another. For example, $\sigma_1 : \pi_0 \xrightarrow{1} \pi_1$ represents one transition from π_0 to π_1. The transition of a VM from one placement status to another placement status correlates to a migration step in the VM migration plan. In a single migration step, either one VM migration (serial) or multiple VM migrations (parallel) can be performed.

Let $\rho(\pi_x)$ represent the safety constraints for the placement status π_x. The safety constraints include capacity, dependency, and security constraints. At each intermediate migration step, safety constraints must be enforced.

The *migration planning problem* is defined as follows: given a set of physical machines, a set of VMs allocated on the physical hosts, the set of VMs to be migrated, the set of hotspot PMs, the set of critical VMs, and the target placement of VMs, find a migration plan (i.e., a sequence of migration steps) that converges the initial placement to the target placement such that the cost is feasible, but for which the capacity constraints of individual hosts, the communication dependency constraints, and the security migration constraints are not violated. Costs may include data storage and transfer, migration interruptions, migration time, and number of migrations.

Formally, we state the problem as follows:

Given:

$$PM, VM, H, M, D, C, R, \pi_0, \pi_\tau$$

Find:

$$\sigma_k \mid \sigma_k : \pi_0 \xrightarrow{k} \pi_\tau \wedge \left(\bigwedge_k \rho(\pi_k) \right)$$

The problem is modeled as a constraints satisfaction problem not as an optimization problem. Our goal is to find a satisfiable migration plan, i.e., one that satisfies all the constraints and thresholds. The solution includes the migration steps required to assemble resources and make the space needed for the target assignment.

4.3.1.1 Virtual Machine Migration Example

We illustrate the VM migration problem using the example configuration depicted in Figure 4.3. For simplicity, we consider only the memory capacity and communication dependency requirements as safety requirements in this example.

Using the same notations presented in §4.3.1, the system is described as follows: $VM = \{VM_1, VM_2, VM_3, VM_4\}$, $PM = \{PM_1, PM_2, PM_3\}$, $M = \{VM_1, VM_2\}$, $D = \{(VM_1, VM_2)\}$, $H = \{PM_1\}$, and $C = \{VM_3\}$. In this example, the communication dependency threshold is set to *one*, $T_D = 1$, which means that any two VMs having dependency between them are required to be allocated on hosts within one link distance from each other. In this example, PM_1 and PM_2 are within one link distance; PM_2 and PM_3 are also within one link distance.

The initial placement state, π_0, is:

$$\pi_0 = \{(VM_1, PM_1), (VM_2, PM_1), (VM_3, PM_2), (VM_4, PM_3)\}$$

The target placement state, π_τ, is:

$$\pi_\tau = \{(VM_1, PM_3), (VM_2, PM_2), (VM_3, PM_2), (VM_4, PM_3)\}$$

The goal is to find a migration plan that makes π_0 converge to π_τ without violating safety constraints. Our framework provides the following migration plan:

1. $\sigma_1 : (VM_2, PM_1) \xrightarrow{1} (VM_2, PM_2)$

2. $\sigma_2 : (VM_1, PM_1) \xrightarrow{2} (VM_1, PM_3)$

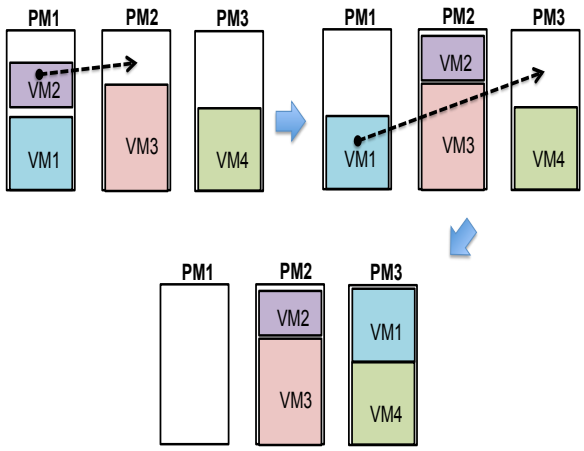

Figure 4.3: An example of VMs migration.

This migration plan maintains the dependency requirements between VM_1 and VM_2 at all intermediate states. Note that migrating VM_1 before VM_2 will break the dependency requirement at the first migration step; PM_1 and PM_3 are within 2 links distance and the threshold is 1. Therefore, this migration sequence is not safe and will be avoided.

This example illustrates importance of the migration sequence to preserving capacity and safety requirements, especially when the size of the problem increases. Building an automated framework to produce migration plans is a great benefit for complex migration situations, including ones that may appear to be infeasible at first glance. Another example that shows the importance of VM migration planning is shown in [33]. This example considers the link bandwidth limits as constraints for VM migration. Our methodology handles both examples (and others) under a common framework.

4.3.2 Framework Overview

Our automatic framework for VM migration planning (see Figure 4.4) is a three stage process. The first stage includes the specification of the initial placement, π_0, of virtual machines in the cloud infrastructure; the specification of the target placement,

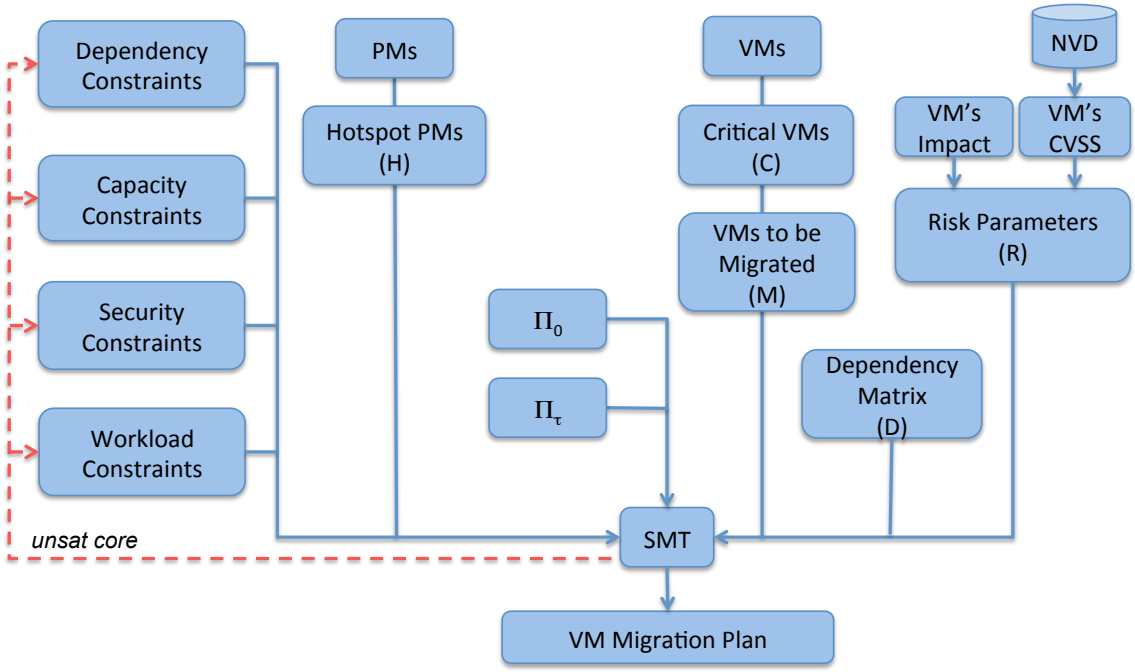

Figure 4.4: VM migration planning framework

π_τ; the specification of hotspot PMs, H; the specification of critical VMs, C; and the calculation of the distance matix, D, and risk parameters, R. Note that the set of VMs to be migrated, M, can be derived from π_0 and π_τ.

Stage two of our framework involves the specification of dependency, capacity, security and workload constraints. These constraints address collocation, bandwidth, security and workload requirements, respectively.

The final stage of our framework for VM migration planning is the generation of a migration plan. In this stage, the various parameters are inputted into the SMT solver. Modeled as a Constraint Satisfaction Problem, the SMT solver then finds a migration plan, represented as a sequence of migration steps, that converges the initial VM placement to the target VM placement with a feasible cost, without violating the dependency, capacity, and security constraints. Costs may include data storage and transfer, migration interruptions, migration time, and the number of migrations.

4.3.3 Migration Planning

In order to generate a VM migration plan that satisfies the dependency, capacity, security and workload requirements, as well as the other framework parameters including risk, we define the following constraints.

Placement Boolean Variable. A decision variable is needed to control the placement of virtual machines on physical machines. This decision variable is restricted to be a Boolean variable as follows:

$$a_{i,j}^{k} \in \{0, 1\} \tag{4.18}$$

The variable $a_{i,j}^{k}$ indicates if the VM_i is mapped to the PM_j at the k^{th} migration step.

Mutual Exclusion. In a cloud environment, while more than one VM may run simultaneously on a single PM, a given VM can only run on one PM at a time.

$$\forall k \quad \sum_{j=1}^{m} a_{i,j}^{k} = 1 \tag{4.19}$$

Constraint 4.19 ensures that for every virtual machine VM_i, there is only one decision variable, $a_{i,j}^{k}$, is set to *one* at every migration step k. The mutual exclusion constraint sets all other decision variables to *zero* once a virtual machine is allocated to a physical machine.

Capacity Limits. The capacity of VMs hosted in a single PM should not exceed the capacity limits of that PM. CPU, memory, disk space, and bandwidth are the main specification dimensions to be used in resources provisioning. Without loss of generality, we consider only CPU and memory dimensions in the current model. Constraint 4.20 represents the *CPU* constraint ensuring that capacity limitations will

not be violated.

$$\forall k, \forall j \in PM \quad \sum_{i=1}^{n} \left(a_{i,j}^{k} * c_{i}^{cpu}\right) < C_{j}^{cpu}, \quad i \in VM \tag{4.20}$$

This constraint limits the total CPU capacity of VMs allocated in a single physical machine not to exceed the CPU capacity of the hosting physical machine. Variables c_i^{cpu} and C_j^{cpu} represent CPU requirement and capacity for VM_i and PM_j, respectively. Similarly, other constraints such as memory, disk space, and bandwidth can be modeled using the same notations used in Constraint 4.20. The generality of the model allows more than one specification dimension to be considered at a time.

Single Migration per VM. Let $\delta_{i,j}^{k}$ represent the change of placement status for VM_i at two consecutive migration steps, k and $k+1$. Formally:

$$\delta_{i,j}^{k} \in \{0, 1\} \tag{4.21}$$

$$(\delta_{i,j}^{k} = 1) \Leftrightarrow (\bigwedge_{q=0}^{k} \neg a_{i,j}^{q}) \wedge (\bigwedge_{r=k+1}^{\tau} a_{i,j}^{r}) \tag{4.22}$$

The Boolean variable, $\delta_{i,j}^{k}$, is set to 1 if the VM_i is migrated from a hosting machine to a different host, PM_j, at the $(k+1)^{th}$ migration step. To avoid unnecessary migrations in which a VM is migrated several times, we limit each VM to be migrated at most one time. Constraint 4.22 ensures this by setting $a_{i,j}^{r}$ equal to *one* for all steps from (k+1) and onward. Also, this constraint helps to prevent situations where the allocation of a VM is flipped in a loop between PMs.

Dependency Constraint. Migrating VMs may break the communication dependency between VMs. Dependent VMs are required to be placed within a predetermined physical distance threshold, T_D. In our framework, we define the distance between physical hosts as number of links between the pair of machines hosting dependent VMs. Enforcing dependency constraints eliminates a primary factor for

performance degradation by maintaining the proximity of dependent VMs. The dependency requirement is given as an input to our framework. A Boolean dependency matrix, D, is defined to encode the dependency between each pair of VMs. Formally:

$$D_{x,y} \in \{0,1\}, \quad x,y \in VM \tag{4.23}$$

$$\forall x \forall y \in D \quad a^k_{x,v} \wedge a^k_{y,w} \rightarrow l_{v,w} \leq T_D \tag{4.24}$$

where $l_{v,w}$ is the physical distance between PM_v and PM_w.

Constraint 4.24 checks the dependency between all VMs at each migration step. In some cases, dependency constraints may result in the migration of VMs that were not scheduled for migration to maintain their dependency distance thresholds, T_D.

Prioritized Migration. The workload characteristics can affect the total migration overhead of all VMs to be migrated [34]. In our framework, we leverage the observations and conclusions that are suggested in the work done in [34]. A VM can be characterized by its workload as follows: *memory-dirty* workload, *disk I/O* load, *NET I/O* load, *CPU* load, and *web server* workload. Each workload affects the migration process to a certain extent. For example: a web server workload VM affects both the migration time and the migration downtime of its *co-hosted* VMs.

To model these observations, each VM is assigned a priority score based on its workload characteristics; thus, the sequence of migrations can be prioritized to decrease the downtime for VMs. VMs are migrated based on their priority score, i.e., the lower the priority score, the higher the migration priority. Let o_x represent the priority score for VM_x. The priority score is given based on the observations provided in [34].

$$\forall x \forall y \in M \quad o_x < o_y \rightarrow \delta^k_{x,v} \wedge \delta^q_{y,w} \wedge k < q \tag{4.25}$$

Constraint 4.25 enforces the migration order of VMs; VMs with higher priority will be migrated before VMs with lower priority. The prioritized migration constraint

provides partial information to the solver about the sequence ordering.

Critical VMs. Some VMs require specific settings and dedicated hardware. Such VMs are considered as critical VMs. Critical VMs are *non-migratable*. Let η_i represent a Boolean variable to flag critical VMs.

$$\eta_i \in \{0, 1\}, \quad i \in VM \tag{4.26}$$

$$\eta_i \rightarrow \sum_{l=0}^{\tau-1} \delta_{i,j}^{l} = 0 \tag{4.27}$$

where $\delta_{i,j}^{l}$ represents the count of how many times a VM is migrated.

Constraint 4.27 asserts the count of migrations for a critical VM to be *zero*. The critical VMs constraint can also be utilized to fix the allocation of VMs that have specific workload; memory-dirty workload for example.

Hotspot PMs Constraint. The set of hotspot PMs, H, identifies PMs scheduled to be switched off. As such, no PM $\in H$ should host a VM at the final placement, π_τ. Let s_j^k indicate the status of a PM at the k^{th} migration step.

$$s_j^k \in \{0, 1\} \tag{4.28}$$

$$\forall j \in H \quad s_j^\tau \rightarrow \sum_{i \in VM} a_{i,j}^\tau = 0 \tag{4.29}$$

Constraint 4.29 ensures that the hotspot physical hosts are excluded at the final migration step, τ, and thus are not hosting any VMs.

Risk Anti-Collocation. The security of a VM in virtualized environments relies on resource isolation among VMs. There is no complete isolation between VMs hosted on the cloud; it is possible to have cross-VM attack from VMs that are collocated on the same host [67]. Thus, during the migration process and before the final allocation of the migrated VMs, VM risk requirements could be violated - e.g., a VM can be allocated intermediately on a PM that hosts other VMs with conflicting risk

requirements.

To provide more secure isolation during the live migration process, we will enforce that no two VMs with conflicting security/risk requirements are hosted on the same PM. For more details about risk calculation, refer to §4.1.5 and §4.1.6. For simplicity, we organize VM risk into three different categories: *high*, *medium*, and *low*. Each PM has a risk tag indicating the type of VMs that it can host according to their risk category. The following constraint ensures that risk requirements are maintained during all steps of the migration plan, including intermediate steps.

$$\forall k \quad a^k_{i,j} \wedge a^k_{x,j} \rightarrow r_i = r_x \quad x, i \in VM \tag{4.30}$$

where r_i is the risk score for VM_i.

Constraint 4.30 ensures that during all migration steps, VMs that are collocated on the same PM have the same risk score.

As previously described, during stage three our framework unwraps all the above constraints to their valid combinations and inputs them into the SMT solver. The solver, in turn, assembles the given combinations and finds a satisfiable assignment, if one exists. In the case where there are no satisfiable solutions, relaxation of constraints may be required. Examples of potential relaxation options included:

- Changing the target placement π_τ: by changing the target placement, it is possible to find a VM migration sequence that leads to π_τ.
- Changing the dependency threshold T_D.
- Allowing VMs to be migrated multiple times.
- Changing the set of critical VMs, C.

Note that the input values such as π_0 and H are asserted as *satisfied* variables. Furthermore, note that the generality of our framework suggests that it can be easily

extended to include other constraints to serve different requirements and applications.

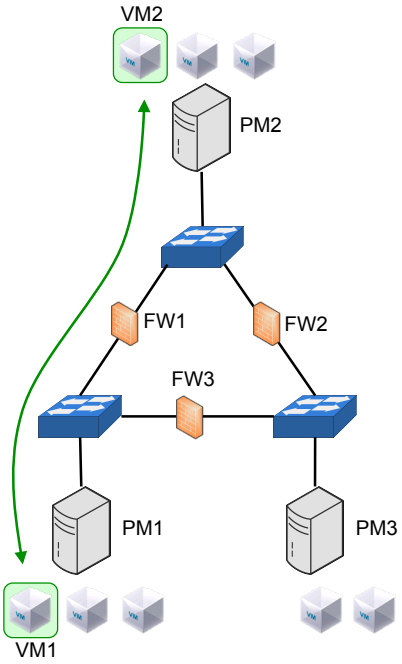

Figure 4.5: VM migration example (before migration)

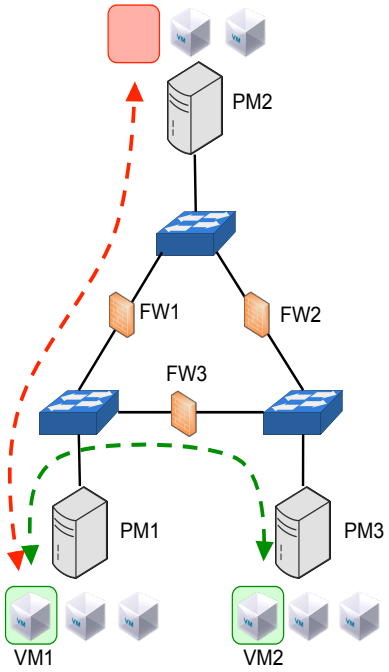

Figure 4.6: VM migration example (after migrating VM_2 from PM_2 to PM_3)

4.4 Virtual Machine Migration Verification

VM migration plays an important role in maximizing resource utilization and performance within cloud computing environments.VM migration, however, results in cloud computing environments whose state is dynamic rather than static, as VMs may be relocated to several different host machines during their life cycles. Migrating VMs between physical hosts changes the deployment of virtual network topologies, which in turn requires changes to physical network middleboxes, such as firewalls and switches, to preserve security and connectivity policies.

It is important to ensure the security/connectivity of all VMs post migration. In other words, the configuration of the virtual network topology under the new physical network deployment must be equivalent to the configuration prior to VM migration in terms of its security and network connectivity policies. The potential for miscon-

figuration of these devices, however, is high, due to inherent configuration complexity, which can lead to policy violations. To illustrate, we provide the following example.

4.4.1 Virtual Machine Migration Verification Example

Figure 4.5 presents an illustrative virtual network deployed to a simple cloud computing environment comprised of three physical hosts and three firewalls. In this example, VM_1 is hosted in PM_1 and VM_2 is hosted in PM_2. Assume for this example that the reachability requirements for this set of VMs allows VM_1 and VM_2 to reach (i.e., communicate mutually) with each other. Therefore, firewall FW_1 is configured to allow network traffic sourced from VM_1 and destined to VM_2, and vice versa.

Assume for performance reasons, however, that the decision is made to migrate VM_2 from PM_2 to PM_3. To maintain an equivalent configuration for the virtual network, all VMs post migration must maintain equivalent reachability requirements relative to the pre-migration configuration. In this simple example, after VM migration, firewalls FW_1 and FW_3 must be reconfigured to: i) block all network traffic between VM_1 and VM_2 via FW_1, traffic that was allowed prior to VM migration; and ii) enable all network traffic between VM_1 and VM_2 via FW_3, traffic that was blocked prior to migration (see Figure 4.6).

4.4.2 Problem Statement

Reconfiguring firewall policies, however, under most circumstances is a non-trivial task. In particular, reconfiguring firewall policies may require the addition, deletion, and/or modification of multiple firewall rules. When multiple VMs are migrated, the complexity can increase significantly. Practice has demonstrated that the identification and modification of these rules are both challenging and prone to error. Incorrect or inconsistent modifications will introduce policy misconfigurations that affect the security policy and reachability requirements of the virtual network. In-

cremental, automated, formal techniques to reconfiguration can provide an efficient and configuration-preserving solution to this challenge. In the following sections we present our approach to meeting this challenge.

4.4.3 Virtual Machine Migration Verification

As previously indicated, cloud network reconfiguration post VM migration may lead to policy misconfigurations – misconfigurations that can jeopardize the security of the cloud computing environment. There are numerous potential sources for misconfiguration such as failure to update properly a network firewall/switch policy or a sequence of policies on a path of network links between old/new VM locations. Thus, proper verification of cloud network configuration equivalency post VM migration is essential to ensure the preservation of cloud security.

Using our model for cloud network configurations, however, formal verification of cloud network configuration equivalency post migration is possible. In particular, cloud network configuration equivalency may be verified via a series of constraints. To define these constraints, consider the following:

- Let VM_i be a migrating VM.
- Let x_i be the location of VM_i *before* migration, and y_i be the location of VM_i *after* migration.
- Let $ARP_b^{(x_i,*)}$ and $ARP_b^{(*,x_i)}$ be the access route policy at location x_i from/to VM_i and all other VMs *before* migration.
- Let $ARP_a^{(x_i,*)}$ and $ARP_a^{(*,x_i)}$ be the access route policy at location x_i from/to VM_i and all other VMs *after* migration.
- Let $ARP_a^{(y_i,*)}$ and $ARP_a^{(*,y_i)}$ be the access route policy at location y_i from/to VM_i and all other VMs *after* migration.

- Let $f^{x_i} \in NF$ be a network firewall that controls network traffic from/to the migrating VM, VM_i, at the old location x_i, i.e. $f^{x_i} \in \{AR^{(x_i,*)} \vee AR^{(*,x_i)}\}$. Similarly, $f^{y_i} \in NF$ is defined to be a network firewall that controls network traffic from/to the migrating VM, VM_i, at the new location y_i, i.e. $f^{y_i} \in \{AR^{(y_i,*)} \vee AR^{(*,y_i)}\}$.

- Let $P_{f_a^{x_i}}$ and $P_{f_b^{x_i}}$ be the firewall policy *after* and *before* migration, respectively.

Given these assumptions, cloud network configuration equivalency may be verified post migration by determining the satisfiability of the following constraints.

Blocked Traffic Constraint. For all migrated VMs, $VM_i \in \{VM_1, ..., VM_n\}$, all accepted traffic at x_i to and from VM_i post migration must be blocked.

$$\forall VM_i \in \{VM_1, ..., VM_n\}. ARP_a^{(x_i,*)} = \phi \tag{4.31}$$

$$\forall VM_i \in \{VM_1, ..., VM_n\}. ARP_a^{(*,x_i)} = \phi \tag{4.32}$$

Similar Traffic Constraint. For all migrated VMs, $VM_i \in \{VM_1, ..., VM_n\}$, all accepted traffic at y_i to and from VM_i post migration must be the same as all accepted traffic at x_i to and from VM_i pre migration.

$$\forall VM_i \in \{VM_1, ..., VM_n\}. ARP_a^{(y_i,*)} = ARP_b^{(x_i,*)} \tag{4.33}$$

$$\forall VM_i \in \{VM_1, ..., VM_n\}. ARP_a^{(*,y_i)} = ARP_b^{(*,x_i)} \tag{4.34}$$

Unaffected Traffic Constraint. For all non-migrating VMs, $VM_j \notin \{VM_1, ..., VM_n\}$, all accepted traffic to and from VM_j must be unaffected.

$$\forall VM_j \in \{VM_1, ..., VM_n\}. \bigwedge_{u,v \neq x} ARP_a^{(u,v)} = \bigwedge_{u,v \neq x} ARP_b^{(u,v)} \tag{4.35}$$

Filtering Rules Removal Constraint. For all network firewalls $f^{x_i} \in NF$ for

all migrated VMs, $VM_i \in \{VM_1, ..., VM_n\}$, must be updated to remove all filtering rules that allow traffic sourced or destined to VM_i.

$$\forall f^{x_i} \in NF, VM_i \in \{VM_1, ..., VM_n\}.$$
$$P_{f_a^{x_i}} = P_{f_b^{x_i}} \setminus P_{f_b^{x_i}}|_{source=x_i \ \vee \ destination=x_i} \tag{4.36}$$

where $P_{f_b^{x_i}}|_{source=x_i \ \vee \ destination=x_i}$ is the restricted BDD after assigning source and destination variables to x_i.

Filtering Rules Addition Constraint. For all network firewalls $f^{y_i} \in NF$ for all migrated VMs, $VM_i \in \{VM_1, ..., VM_n\}$, must be updated to include all filtering rules that allow traffic sourced or destined to VM_i.

$$\forall f^{y_i} \in NF, VM_i \in \{VM_1, ..., VM_n\}.$$
$$P_{f_a^{y_i}} = P_{f_b^{y_i}} \bigwedge P_{f_b^{x_i}}|_{source=x_i \to y_i \ \vee \ destination=x_i \to y_i} \tag{4.37}$$

where $P_{f_b^{x_i}}|_{source=x_i \to y_i \ \vee \ destination=x_i \to y_i}$ is the restricted BDD after assigning source and destination variables to x_i and then replacing x_i with y_i. In other words, the filtering rules that are removed by Constraint 4.36 will be updated to reflect the new location, y_i, and then added to the firewall policy.

4.4.4 Auto-Policy Reconfiguration

In the previous section, we presented an approach for modeling a cloud network configuration as a set of access routes and a formal approach for verifying the correctness of post VM migration cloud network configurations. In this section we present an automated approach to reconfigure firewall policies post VM migration.

Our approach to cloud network reconfiguration is unique in that it is incremental. Consequently, it offers an effective alternative to a complete regeneration of firewall policies from reachability requirements and security policies for all network switches

and firewalls.

There are two main drawbacks to the complete regeneration approach. The first drawback is one of efficiency. Every time a VM migration is performed, the entire cloud network configuration must be generated, including for those devices that are unaffected by the migration. The second drawback is that the new cloud network configuration requires an additional, complete verification to ensure the proper translation of reachability requirements and security policies.

Our incremental reconfiguration approach leverages the prior cloud network configuration and introduces modifications to update selected network switches and firewalls. In particular, we process each affected firewall policy, represented as a BDD, by removing all network flows that are related to the migrating VMs at the old location and leaving the remaining non-migrating VMs network flows. The resulting BDD is the new firewall policy. The removed network flows are then added to the firewall policies that control the migrated VM at the new location. Then, we translate this BDD into a set of firewall rules by traversing all paths from the root to the terminal node.

Our approach leverages previous work on building and verifying network configurations [42]. *ConfigLEGO* is an imperative framework that provides an open programming platform for building and analyzing the entire network security configuration both globally and systematically based on user requirements [42]. The *ConfigLEGO* framework provides a C/C++ API as a software wrapper on top of a BDD engine to allow users to define network topologies and configurations and then perform verification and diagnostic analyses at various abstraction levels, without requiring knowledge of BDD representation or operations.

To develop our incremental reconfiguration approach, we added new features and functions to *ConfigLEGO* to enable the framework to deal with virtual environments. An extra level of reachability analysis is defined to verify VM reachability and VM

Algorithm 1: Auto Firewall Policy Generation

```
Data: devices configuration
Data: network topolgy
Result: Updated firewall policies affected by VM
        migration
1  create and initialize network;
2  add network devices;
3  connect network devices;
4  assign VMs to hosts;
5  build devices BDDs;
6  build network BDD;
7  migrate VMs;
8  for every migrated VM (v) do
9      get r = accessible route (any VM, v);
10     for every firewall (f) in accessible route (r) do
11         extract network flows between (any VM, v) from
           firewall f BDD;
12         translate firewall f BDD into firewall rules;
13     end
14 end
```

Figure 4.7: Auto firewall policy generation

migration.

Figure 4.7 shows the algorithm that summarizes the general steps performed to reconfigure incrementally firewall policies affected by VM migration.

The following *ConfigLEGO* code illustrates how we update the firewall policies after VM migration. The cloud network and VM migration used in this example is the example shown in Figure 4.5 and Figure 4.6.

First, the network is initialized:

```
Network N;
```

Second, all devices are added to the network. Each type of device has a given configuration file. The configuration file contain information about each object such

as: IP address, open ports, firewall policy, routing table, etc.

```
Host PM1("h1.txt");
VM V1("v1.txt");
Firewall FW1("f1.txt");
Router R1("r1.txt");
```

After adding network devices, the network links are defined. For example, *PM1* and *R1* are connected as follows:

```
N.link(PM1, ANY_IFACE, R1, 1);
```

This statement will link *PM1* through any interface to router *R1* through interface "1" in network *N*.

After linking all components in network *N*, a BDD for the network is generated – first by generating a BDD for each device, then by generating a BDD for the entire network. This task is acomplished by invoking the statement:

```
N.buildDeviceBDD();
N.buildGlobalBDD();
```

After building the network, we calculate the accessible routes, e.g., from VM_1 and VM_2 or $AR^{(VM_1,VM_2)}$ in the example. Knowing the access routes will provide the list of network devices that are affected by VM_2 migration. The following code filters all firewall devices in $AR^{(VM_1,VM_2)}$:

```
N.getPathObjects(src, dst, fwVec, FIREWALL);
```

In this code, the getPathObjects(...) function returns a vector, $fwVec$, of all firewall objects between a source domain src and a destination domain dst. Each firewall policy in the accessible route $AR^{(VM_1,VM_2)}$ is updated by extracting all network flows between VM_1 and VM_2 from its BDD. This step is done using the "restrict(...)" function. The restrict function sets the variable values in a BDD as specified in the given parameters. The following restrict example, sets all source IP address values in BDD $FW1$ to the value (10.11.12.13).

```
restrict(FW1, SRC_IP, "10.11.12.13");
```

After restricting the firewall BDD and removing network flows affected by the VM migration, the resulting BDD represents the firewall policy post migration. To translate a BDD into a set of firewall rules, we use "satone(...)" function. This function returns a satisfiable variable assignment in a given BDD. A satisfiable assignment represents a firewall rule. Then, we continue extracting all satisfiable assignments, one by one, until we traverse the entire BDD.

```
bdd r = satone(FW1);
```

However, at this point, the generated firewall rules are not ordered in the optimal order. The optimal rule ordering places the rules with high matching rate at the top. To obtain the best results, we update the rule ordering by comparing matching rates in the firewall log files.

The result is a new network cloud configuration, one that is a reconfiguration of the previous cloud network configuration.

4.5 Software Defined Networks Misconfigurations

The SDN paradigm provides a centralized, programmable, global view of a network environment. Abstracting network control from the underlying physical network is

attractive to cloud infrastructures as SDNs can reduce management overhead and provide additional benefits by easing cloud provider efforts to monitor network usage, provide network isolation, manage network configuration, provision infrastructure resources, and plan for virtual machines migration.

Among several protocols used to run and implement SDNs, OpenFlow is a very prominent, open standard. The OpenFlow standard separates the data plane and control plane [46, 48], the two elements that form the core of a SDN. By separating the data and control plans, cloud providers leverage the OpenFlow controllers to install filters (e.g., match, count and action) on OpenFlow switches to manage in a centralized and programmable manner the flow and processing of data across the network. We use the OpenFlow protocol to characterize the problem of SDN misconfiguration and illustrate the power of our formal methods-based framework to mitigate such misconfigurations.

4.5.1 Problem Statement

SDNs, like any other network architecture, are susceptible to network misconfigurations. There are two main components in any SDN, the controller and the forwarding switches. The controller inserts, modifies, and removes filters in switches in order to enforce network-wide policies or properties (e.g., guests should access the internet only through a proxy) for data flow and processing [46].

It is assumed that the integrated behavior of the installed filters will implement the desired network policies and properties. However, the complexity of these policies and properties, particularly in the aggregate, can give rise to the following situations: (1) the presence of a semantic gap between the controller platform (e.g., NOX [48]) and the filter tables in the data processing units; (2) the distribution of access control that supports aggregate flows (wildcards) and many different actions; (3) the ability for different users to share one controller; and, (4) the ability to use multiple controllers within the same domain. These situations individually as well as collectively increase

the potential for both intra-federated (single domain) OpenFlow configuration conflicts and inter-federated conflicts (multi-domain) OpenFlow configuration conflicts. The result can lead to a violation of end-to-end policy enforcement.

Our formal methods-based framework attempts to address these situations by: (1) encoding OpenFlow configurations using BDDs to disambiguate policies through the use of a priority-based matching semantic to manage competing actions; (2) constructing a unified model of the global behavior of an OpenFlow SDN in the presence of multiple controllers and users; and, (3) providing a generic policy verification interface using BDD-based symbolic model checking and temporal logic of the OpenFlow SDN configuration.

Our framework therefore can be used to: (a) verify the consistency of different switches and controllers across federated OpenFlow infrastructures; (b) validate the correctness of the configuration synthesis; (c) debug reachability and security problems; and, (d) assess the consistency of SDN policies. Our framework can also be used as a foundational methodology to conduct "what-if" analysis to study the impact of the new SDN network configurations by simply changing the state in the FlowTables and then analyzing the effects. For cloud infrastructures, SDNs provide valuable support for virtualization by making cloud management simpler and, under our framework, less susceptible to misconfigurations.

4.5.2 Framework Overview

Our framework for SDN configuration analyses (see Figure 4.8) involves two stages. Stage one includes the specification of OpenFlow SDN rules and policies as well as the establishment of rule prioritizations. Stage two involves query specification and validation. The set of valid queries is defined to include any well-formed temporal logic formula.

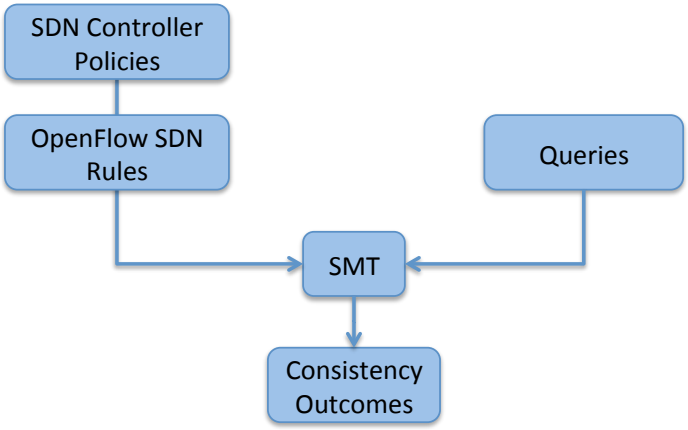

Figure 4.8: SDN configuration analysis framework

4.5.3 Formal Modeling of OpenFlow SDN Configurations

OpenFlow SDN configurations are collections of policies, each defined by a sequence of rules that are deployed by controllers within the OpenFlow switches. Each rule acts, when applicable, as a filter that affects data flow and processing. As such, the terms "rule" and "filter" are often used interchangeably in this context. Formally, a rule is defined as follows:

Definition 3 *An* OpenFlow SDN Rule*, R_i, consists of a set of constraints on a set of k filtering fields, $F = \{fv_1, fv_2, \ldots, fv_k\}$, together with an action, a_i, from the set of all actions, A.*

Each rule can be written in the form:

$$R_i := C_i \rightsquigarrow a_i$$

where C_i is the constraint on the filtering fields that must be satisfied in order for the action $a_i \in A$ to be triggered. The condition C_i can be represented as a Boolean expression over the filtering field values $fv_1, fv_2, \ldots, fv_k$ as follows:

$$C_i = fv_1 \wedge fv_2 \wedge \cdots \wedge fv_k$$

Example field values include IP addresses, port numbers, user IDs, and controller IDs. An OpenFlow SDN policy, therefore, is defined as a sequence of rules (or filters). More formally:

Definition 4 *An* OpenFlow SDN Policy, $P = R_1, R_2, \ldots, R_n$, *is a sequence of* n *rules that determine the appropriate action performed on any incoming packet.*

A priority-based matching semantic is applied to the FlowTable at each OpenFlow switch. This semantic is equivalent to the first-matching semantic of firewalls since the first rule can be reduced to the second rule by simply ordering the rules based on their priorities. We assume that if two rules have the same priority the first match will take precedence. Therefore, the FlowTable matching semantic fits very well with the If-else Normal Form (INF) and can be encoded using BDDs as follows:

$$P_a = \bigvee_{i \in index(a)} (\neg C_1 \wedge \neg C_2 \ldots \neg C_{i-1} \wedge C_i) = \bigvee_{i \in index(a)} \bigwedge_{j=1}^{i-1} \neg C_j \wedge C_i \tag{4.38}$$

Such that $priority(C_{i+1}) < priority(C_i)$, and $index(a) = \{i \mid R_i = C_i \rightsquigarrow a\}$

Formally, the representation of the entire FlowTable (for all actions, users and controllers) for switch j is defined as follows:

$$P(j) = \bigvee_{\forall n \,=\, a_i \,\in\, Action} P_n. \tag{4.39}$$

Let us assume that user and controller IDs are encoded as Boolean variables as u and c, respectively. Under this assumption we can represent the FlowTable for filters of action a created by user u on controller c as follows: $P_a^{u,c} = P_a|_{u,c}$ where $|$ is a restrict operation in the BDD. Therefore, the representation of all FlowTables in the federated network N that belongs to user u and controller c can be defined as:

$$\phi^{u,c}(N) = \bigvee_{\forall j = switch \in N} \; \bigvee_{\forall n \,=\, a_i \in\, Actions} P_n^{u,c}(j). \tag{4.40}$$

Assuming that the default action when a traffic flow does not match any of the filters at a switch is to encapsulate and forward to the controller, then this traffic space that represents these flows will be $\neg P(j)$.

4.5.4 Query Examples

Our framework provides a generic policy verification interface for symbolic model checking of the OpenFlow SDN configuration. Any temporal logic formulae can be used. In this section, we classify the types of verification experiments that can be performed on multiple OpenFlow infrastructures. We also present examples of important properties to be verified in an OpenFlow network.

Intra-Federated Consistency Verification (Intra-Federated Flow Isolation). As the OpenFlow protocol allows for wildcard (flow aggregation) of access control configuration, the interdependency between different access control policies installed by the same user over time, or multiple users working on the same or different controllers might conflict with each other causing errors in the OpenFlow operation such as reachability problems or a security violation. General examples of this type of conflicts (e.g., shadowing, correlation) are described in [68, 35, 69]. Examples within an OpenFlow environment are as follows (refer to § 4.5.3 for notation):

Example (1): "User flow spaces should be completely disjoint/isolated," formally:

$$\bigvee_{j=1}^{n-1} \bigwedge_{i=j}^{n-1} P^{i,C} \wedge P^{i+1,C} = FALSE \tag{4.41}$$

such that users $i, j \in Controller(C)$.

Example (2): "All production (non-experiment) flows should be forwarded normally," formally:

$$\Psi \rightarrow \bigvee_{j=1}^{n} P_{a_i}(j) = TRUE \tag{4.42}$$

such that Ψ is a BDD that represents the production traffic flows and a_i equals the

$normal\_forward$ action. Notice that this does not impose any restriction on non-production traffic.

Inter-Federated Consistency Verification (Inter-Federated Flow Isolation). This query explores inconsistencies between any two rules within FlowTables across the network. For example, "FlowTables contained in switches with different controllers within the same domain exhibit consistent action for the same overlapping flows". This can be formally defined as follows:

$$\bigvee_{j=2}^{n-1} \bigwedge_{a_i \in Actions} \bigwedge_{j=1}^{n-1} P_{a_i}(j) \wedge \neg P_{a_i}(j+1) \;=\; FALSE \tag{4.43}$$

such that the switch $j \in Controller(C)$.

Intuitively, this means that no two FlowTables will execute filters of different actions on the same traffic flow. This eliminates both rule shadowing and spuriousness between switches in the path [36]. It is worth mentioning that this property might only be used with actions that require path consistency/stability, such as forward, limit, and QoS-classes, but not necessarily for any general action such as encrypt that can be performed by certain switches.

Property-based Verification for Inter-federated Flow. Using model checker technique like in ConfigChecker [4], our framework can verify general network properties using temporal logic. Some OpenFlow examples are presented in the following:

Example(1): "If a packet is ever encrypted, it will eventually reach the destination as plain text", formally:

$$Q = (src = a_1 \wedge dest = a_2 \wedge loc(a_1) \wedge encrypt \rightarrow AF\ decrypt \wedge loc(a_2)) \tag{4.44}$$

where encrypt/decrypt are flags to mark whether a packet is encrypted or decrypted and $loc(a_1)$ indicates the current location of a packet is at address a_1.

Example(2): "Guests in the network can only access the Internet through a proxy

server" [46], formally:

$$(loc(a_1) \ \wedge \ src = a_1 \ \wedge \ guest(a_1) \ \wedge \ dest = a_2 \ \wedge \ Internet(a_2)) \\ \rightarrow \neg EF(src = a_1 \ \wedge \ dest = a_2 \ \wedge loc(a_2) \ \wedge \ \neg proxy) \quad (4.45)$$

In this example, $guest(a_1)$ means that the source address a_1 is used by the guest, $Internet(a_2)$ means that the destination address a_2 is outside the domain, and $proxy$ is a flag to indicate if a packet is sent through a proxy server or not. Literally, if a packet is located at an IP address used by a guest and it is destined to an address outside the domain, then the packet should pass through a proxy server.

Example(3): "VoIP phones are not allowed to communicate with laptops" [46], formally:

$$(loc(a_1) \ \wedge \ src = a_1 \ \wedge \ VoIP(a_1) \ \wedge \ dest = a_2 \ \wedge \ laptop(a_2)) \\ \rightarrow \neg EF(src = a_1 \ \wedge \ dest = a_2 \ \wedge loc(a_2)) \quad (4.46)$$

In this example, $VoIP(a_1)$ means that a_1 is assigned to a VoIP phone; similarly, $laptop(a_2)$ indicates that a_2 is a laptop machine.

Flow Isolation. The configurations given to a user describe the flows that should be isolated from the other user flows. This means that the flows cannot be shared by other users. Assuming S_i and S_j are BDDs for different flows, we can formally verify this as follows: $S_i \ \cap \ S_j \ = \ \phi$, such that $i, \ j \ \in \ Users$.

Flow Actions. Our framework responds to a message m from a controller by: allowing it if m matches the user configuration; rewriting it if m does not match the user slice but it can be restricted to match it; or, dropping it if m does not match

and cannot be restricted to match it. This can be formally checked as follows:

$$((m \rightarrow S_i = TRUE) \rightarrow allow)$$
$$((m \rightarrow S_i = FALSE) \rightarrow drop)$$
$$((m \rightarrow S_i = Exp) \rightarrow rewrite)$$

where m is a message, S_i is the policy for user i, and Exp is a Boolean expression.

Interactive Debugging via Counter Examples. For each one of these examples, if the property was not satisfied, our framework can present a counter example (i.e., misconfiguration) that would invalidate this property in the network. This can be used for interactive debugging by fixing and changing the configuration iteratively until the property is satisfied.

4.5.5 FlowTable Pipeline Misconfiguration

The OpenFlow protocol permits the specification of multiple FlowTables within a single network switch. The multiple FlowTables are collectively referred to as a *pipeline*. In other words, an OpenFlow pipeline is composed of multiple "stops" where at each stop various tasks are performed. The effect of a pipeline is to have multiple FlowTables performing multiple actions on a packet at a given network switch.

OpenFlow defines mechanisms to constrain pipeline specifications. For example, in a pipeline, FlowTables at each switch are numbered starting from 0. A network packet is matched against rules in table 0 first, then forwarded to next table in increasing order. OpenFlow packet forwarding from one table to another if the table ID of the destination is larger than current table ID, i.e., backward flow is not allowed. Network packets are forwarded from one table to the next within a pipeline using the "goto table" command. OpenFlow rules may also change the status of a packet via the "set" operation. Having multiple "goto table" commands in different FlowTables at a single switch creates several pipelines and can make detecting policy misconfigurations

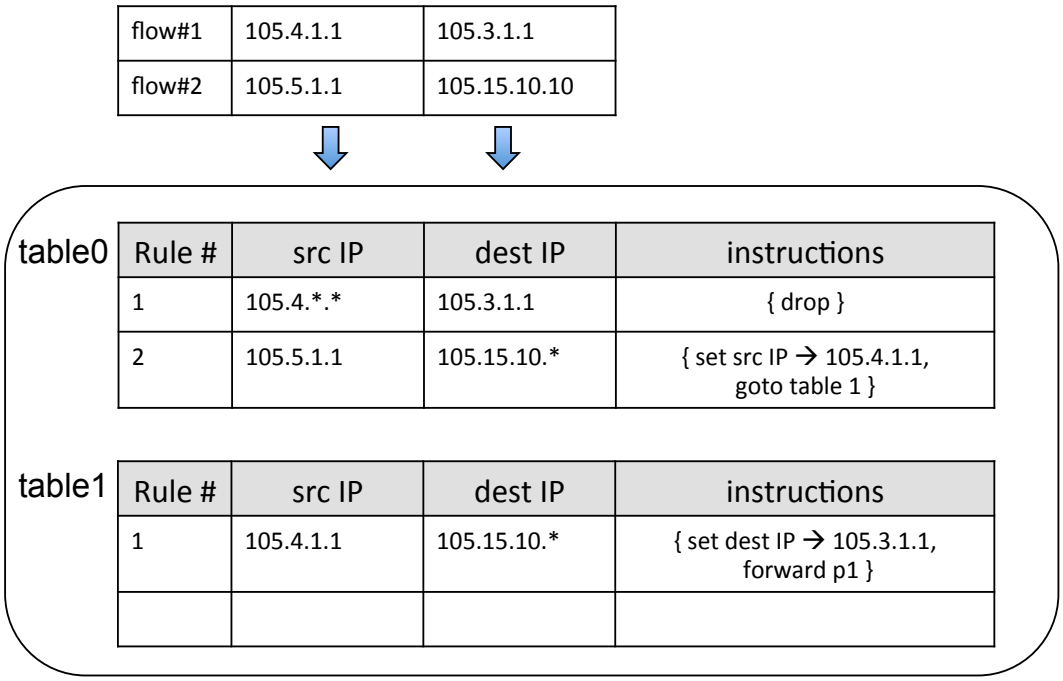

Figure 4.9: Example to illustrate FlowTable pipeline misconfigurations caused by "set" and "goto table" actions.

a complex task. The presence of "set" operations in FlowTable rules can make it difficult to track updates and verify compliance with desired security policies. As such, the dynamic nature of the SDN architecture and the presence of complex rules that can change packet headers, such as "set" and "goto table" commands, create an environment that is rich for misconfigurations.

Our framework models the aggregate behavior of all FlowTables in a single switch using BDDs. Our framework also can verify and detect misconfigurations in all possible pipelines that result from using the "goto table" command in FlowTables.

To understand better how the power and complexity of SDNs can lead to FlowTable pipeline misconfigurations, we offer the following example. This example, shown in Figure 4.9, illustrates how the "set" command can lead to a misconfiguration of an OpenFlow pipeline. Let us assume that the desired network flow policy should block all network traffic coming to (105.3.1.1) from subnet (105.4.*.*). In network FlowTable table0, rule#1 implements this policy and blocks all traffic from (105.4.*.*)

to (105.3.1.1). When a network packet is matched against this rule, rule#1 in table0, the packet will be dropped. Thus, when packet representing incoming flow #1 arrives at the network switch and is processed by Flow Table table0, the packet will be dropped and the network flow policy will not be violated.

The packet representing incoming flow #2 coming from (105.5.1.1) to (105.15.10.10) arrives at the switch, it also will be processed in the pipeline starting from table0. This packet, however, will not match rule #1 in table0; therefore, it will be matched against the next rule, rule #2 in table0. In this situation, the packet matches the rule header and the associate actions are triggered. There are two actions in rule #2. The first action includes a "set" command that modifies the source IP of the packet to (105.4.1.1). This modification does not affect the matching of rule #2 - the header need only be matched successfully once prior to the execution of the action(s). The second action includes a "goto table" command that forwards the "modified"network flow to table1 in the pipeline. In table1, the packet will match rule #1 and the associated actions will be activated. The "set" action will modify the dest IP to (105.3.1.1) and then the packet is forwarded. This example illustrates how multiple "set" and "goto table" actions can be used to modify network flows in ways that violate desired network policies. Such situations represent a misconfiguration of the SDN, yet they are not readily obvious to the end user.

FlowTable Pipeline Verification. Using CTL queries, we can write the following query to detect FlowTable pipeline (intra-pipeline) misconfiguration that is discussed in the example above:

$$\begin{aligned}[(loc(switch(j)_b) \ \wedge \ src = a_1 \ \wedge \ dest = a_3 \ \wedge \ drop(a_1, a_2) \\ \rightarrow \neg EF(loc(switch(j)_a)) \ \wedge \ src = a_1 \ \wedge \ dest = a_2\end{aligned} \tag{4.47}$$

where $loc(switch(j)_b)$ and $loc(switch(j)_a)$ represent the actual location of the packet

before the switch and after the switch, respectively. The function $drop(a_1, a_2)$ means that there is a rule implemented at the switch to drop flows between a_1 and a_2.

To capture intra-pipeline misconfiguration at a switch, we need to detect all modifications by the "set" and "goto table" actions that cause conflict with the actual implemented policy.

Intuitively, equation 4.47 means that there should not be any modification on the flow at the FlowTable pipeline in which the result is different from the encoded rules (i.e., the flow destination should not be modified from a_3 to a_2 when the flow source is a_1 and there is a rule drops flows between a_1 and a_2).

4.5.6 Soundness and Completeness for OpenFlow Configurations

Using CTL queries, we can verify the soundness and completeness of OpenFlow configurations. We do not assume all flows can be targeted from any source IP to any destination IP, even if there is a physical connectivity between IP addresses. The valid connectivity is determined based on the Connectivity Requirement Policy (CRP), which considers the authorization access of all FlowTables users. Let us define $FlowConnect(src, dst)$ as a characterization function for all sets of allowed flows from src to dst that represent CRP.

Definition 5 *A configuration, C, is sound if, for all nodes u and w, all possible paths from u to w are subset of (or implies) authorized paths in FlowConnect(u, w).*

Intuitively, no node u can communicate with node w using C if this is not allowed in the CRP. Formally, this soundness property can be written as the following CTL query:

$$[(loc(a_1) \ \wedge \ src = a_1 \ \wedge \ dest = a_2 \ \wedge \ EF(loc(a_2)) \rightarrow FlowConnect(a_1, a_2)) \quad (4.48)$$

Definition 6 *A configuration, C, is complete if, for all nodes u and w, FlowCon-*

nect(u, w) contains all possible paths from u to w.

In other words, if u is allowed to communicate with w in CRP, then there must be a path from u to w to allow this communication. Formally,

$$FlowConnect(a_1, a_2) \rightarrow [loc(a_1) \wedge src = a_1 \wedge dest = a_2 \rightarrow EF(loc(a_2))] \tag{4.49}$$

4.6 Summary

In this chapter, we addressed and investigated some challenges of secure and manageable cloud computing infrastructure services. A formal-method based approach composed of five frameworks was presented to address these challenges. We summarize these frameworks as follows.

In §4.1, we addressed the problem of managing VMs access control lists. By utilizing the traffic similarity of VMs, VMs are grouped into groups in which each group is managed by a single ACL. The presented framework used formal-method based approach to encode the distance between VMs using BDDs. The groups are synthesized using a SMT solver. The automatic creation of VMs groups reduces the potential of having policy misconfigurations by reducing the number of ACLs required to manage all VMs by sharing ACLs.

In §4.2, we addressed the problem of secure resource allocation. The presented formal-method based framework considers security requirements at the decision time to allocate resource. The framework reduces the residual risk that result from allocating VMs that have conflicting security requirements on the same host. It also reduces the cost that result from post-security decisions that are needed to solve conflicting requirements.

In §4.3, we addressed the problem of preserving capacity, dependency, and security requirements during VM migration. To meet the challenges of preserving security and performance requirements while VMs are migrated, we presented a formal-method

based framework to find the sequence in which VMs need to be migrated in order to preserve security and performance requirements. The migration plan framework models security and performance requirements as constraints satisfaction problem and relies on SMT solver to find the migration sequence.

In §4.4, we addressed the problem of verifying the equivalence of cloud network configuration before and after VM migration. We presented a formal-method based framework to verify that the updated cloud network configuration is equivalent to the configuration prior to VM migration in terms of its security and network connectivity policies.

Moreover, we addressed the problem of reconfiguring cloud network after VM migration due to the changes of virtual network topologies. Incorrect or inconsistent modifications will introduce policy misconfigurations that affect the security policy and reachability requirements of the virtual network. We presented an incremental, automated, and formal technique to reconfiguration to provide an efficient and configuration-preserving solution to this challenge.

In §4.5, we addressed the SDNs misconfigurations. We argued that cloud datacenters will utilize SDNs to provide better manageability. Before adopting SDNs to host cloud solutions, the risk of SDN misconfigurations needs to be addressed. We presented a formal-method based framework to detect inter/intra policy misconfigurations and FlowTable pipeline misconfigurations in SDNs.

The presented frameworks use formal-method based approaches to provide solutions to the discussed challenges. Across these frameworks, we use BDDs to model policies and ACLs, and constraint satisfaction modeling to encode satisfiability problems.

CHAPTER 5: IMPLEMENTATION AND EVALUATION

In the previous chapter we presented a unified set of frameworks under our formal methodology designed to make IaaS cloud environments more secure and cloud configuration management much simpler. In particular we presented a formal methods-based approach to provide better solutions to the following problems: 1) creating and managing access control lists for virtual machines running in IaaS cloud environments; 2) allocating resources to virtual machines in a manner that considers both security and risk at the time of provisioning; 3) migrating virtual machines from one host to another in a manner that does not violate security requirements; 4) verifying that cloud network configuration is preserved after migration and reconfiguring the cloud network to reflect the new state after VM migration; and, 5) detecting policy misconfigurations in Software Defined Networks. In this chapter, we present the implementation and evaluation methodology for each aspect of our approach as well as discuss evaluation results.

5.1 Managing Access Control Policies for Virtual Machines

We have presented a semi-automatic, formal methods-based framework for creating security groups and managing ACLs for VM security groups. Under our framework, VMs are group based on their incoming traffic similarity. The framework allows VMs to be grouped whose incoming traffic requirements are not identical. A threshold is defined to limit the degree of difference among reachability requirements that is permitted for each VM security group. After grouping VMs, the framework generates (i) an ACL for each security group, and (ii) a set of iptable rules for each VM to manage the variance among reachability requirements within the VM security group.

The proposed framework reduces the likelihood of conflicting ACLs and/or misconfigured ACLs. Furthermore, each VM security group serves as logical zone designed to reduce residual risk for VMs. In the following, we present experimental evaluation of our framework. We begin by describing the experimental setup and conclude with a summary of results.

5.1.1 Experimental Setup

In this experiment, we compare our framework for the formation of VM security groups with the Amazon EC2 grouping model. All evaluation results were simulated on 3.10 GHz quad core CPU with 8 GB memory. For SMT formalization, we used Yices 1.0.36 SMT solver [12] to encode the presented constraints.

The framework is implemented in C++ and all constraints are asserted by invoking Yices C API [12]. All input data and thresholds for the framework are provided through configuration files, which are then parsed and translated into Yices constraints using API functions provided by the solver. The following example shows how to translate an instance of constraint 4.2, namely s_{12}, into an assertion:

```
yices_expr args[2];
args[0]=yices_mk_eq(ctx, s[1][2], zero);
args[1]=yices_mk_eq(ctx, s[1][2], one);
yices_assert(ctx, yices_mk_or(ctx, args, 2));
```

After asserting all constraints, the encoded problem instance is checked using the following Yices API function:

```
yices_check(ctx);
```

The parameter `ctx` is the context, which stores a collection of declarations and assertions. If a satisfiable assignment is found, the output displays the values that satisfy for the asserted variables.

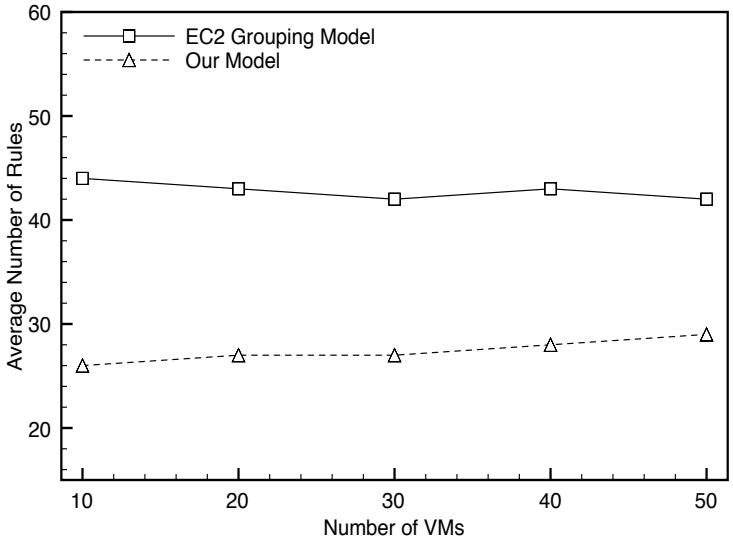

Figure 5.1: The average number of rules required to write a policy for two cases: EC2 grouping model and our grouping model.

5.1.2 Results

We evaluate our framework both in terms of manageability and risk reduction. Manageability is evaluated in terms of the number of rules required to produce the VM security groups. Risk reduction is evaluated in terms of the average VM residual risk.

Manageability. In this round of experiments, we measure manageability based on the average number of rules required for writing an ACL policy for all VMs. We used the following setup to evaluate the manageability. A set of 100 different flows is created randomly, each VM has an average of 50 flows to define the reachability requirements, $\mathcal{T}$=25, and 10%-20% of the VMs are set to be similar. Figure 5.1 compares the average number of rules required to write a policy for our approach against the EC2 model in which only the similar VMs are grouped together. Without grouping, the average number of rules per VM is 50. The average number of rules decreases according the similarity percentage between VMs using EC2 model. The results show that our approach achieves better manageability in term of average

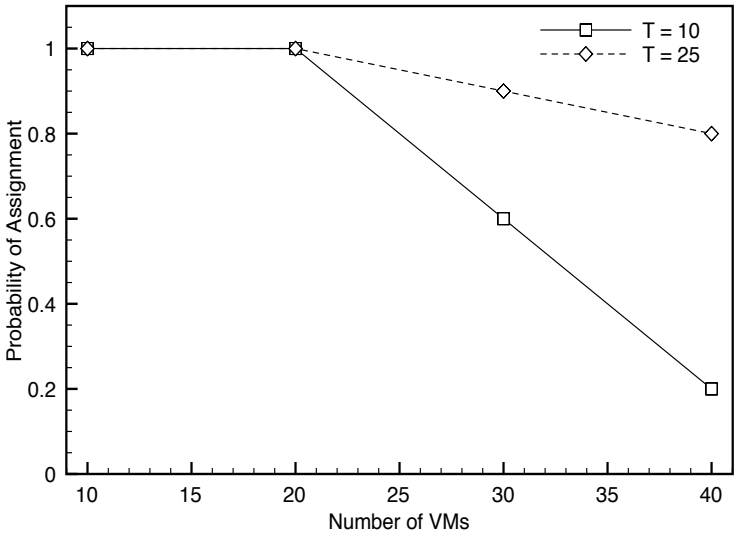

Figure 5.2: The effect of $\mathcal{T}$ on the probability of finding a security group memberships assignment.

number of rules required to implement the policy. In this experiment, we used the same rule format to write ACLs for our grouping model and Amazon EC2 model. A rule in the ACL contains five fields: protocol number, source IP address, source port number, destination IP address, and destination port number.

We also assessed the effect of the maximum number of iptable rules per VM, $\mathcal{T}$, on the probability of finding a satisfiable security group membership assignment. Figure 5.2 shows the evaluation for two cases: $\mathcal{T} = 10$ and $\mathcal{T} = 25$. The probability of finding an assignment decreases dramatically as $\mathcal{T}$ threshold decreases and number of VMs increases.

Risk Reduction. We evaluate the average risk introduced in the system for four cases: (1) no grouping; (2) grouping using the EC2 security group model; (3) grouping using firewalls only; and (4) our model which includes security groups and applying all risk reduction countermeasures. In the no grouping case, we assume that the VMs have no security restrictions to communicate with each other. Thus, we set the reachability variable b_{xy} to 1 for every pair of VMs. In the case of the EC2 model, the reachability is decided at the security group level. The reachability

variable for our model's case is set based on the reachability requirement. We assigned reachability requirements for the VMs randomly, based on Boolean reachability (i.e., two VMs are either reachable or not reachable). In the case of our model, we used the following rates for various risk reduction countermeasures: *allow*(0%), *deny*(100%), *inspect*(80%), *encrypt*(50%), and *anti-collocate*(100%). We assumed that security services are provided by aggregator switches [70].

In order to perform a proper comparative analysis, we need to adjust our calculation of VM residua risk (see §4.1.6) to account for the test case where security groups are not employed. As such, the risk metric for VM_x is calculated as follows for comparative analysis.

$$r_x = \frac{\sum_{y=1}^{n} b_{yx} * v_y}{\sum_{y=1}^{n} v_y} * I_x \tag{5.1}$$

where r_x is the risk for VM_x, b_{yx} is the reachability between VM_y and VM_x, v_y is the vulnerability score for VM_y, and I_x is the impact score for VM_x. Each VM is randomly assigned a vulnerability score in the range [0.01-0.99] and an impact score in the range [1-100]. Figure 5.3 shows the average risk for the four cases. Compared to the no grouping case, our model (showed by the line graph labeled "Security Enforcement") achieved around 64% risk reduction, while the EC2 model achieved around 11% risk reduction.

5.2 Security-aware Resource Allocation in Clouds

We have presented a framework for security-aware resource allocation. This framework builds on our framework for VM security group assignment. Our objective is to find a resource provisioning that satisfies both the customer requirements and the cloud provider requirements. The solution will not necessarily be the optimal one, but it is guaranteed to satisfy all input constraints if such a solution exists. We contend that our framework increases the likelihood finding a satisfiable provisioning. We demonstrate the effect of the number of security-enabled switches on the probability

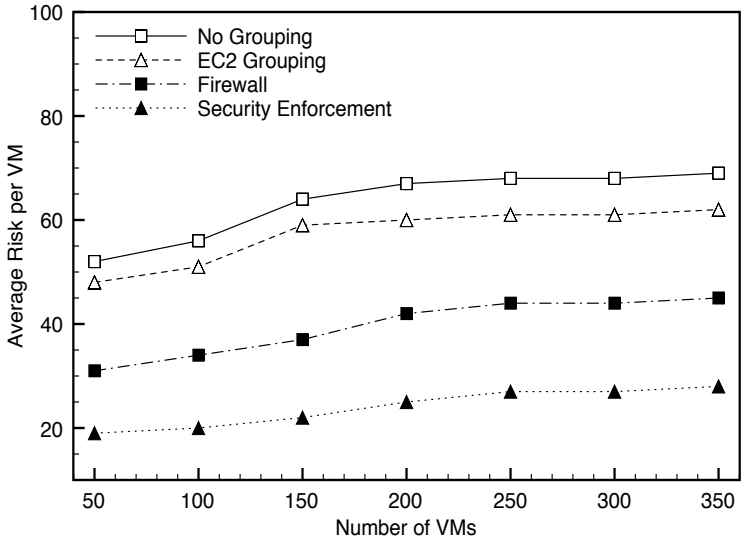

Figure 5.3: The average risk per VM for four cases: no grouping, EC2 grouping model, firewall filtering, and our framework.

of identifying a satisfiable solution. We also evaluate the performance characteristics of our approach. In the following, we begin by describing the experimental setup and conclude with a summary of results.

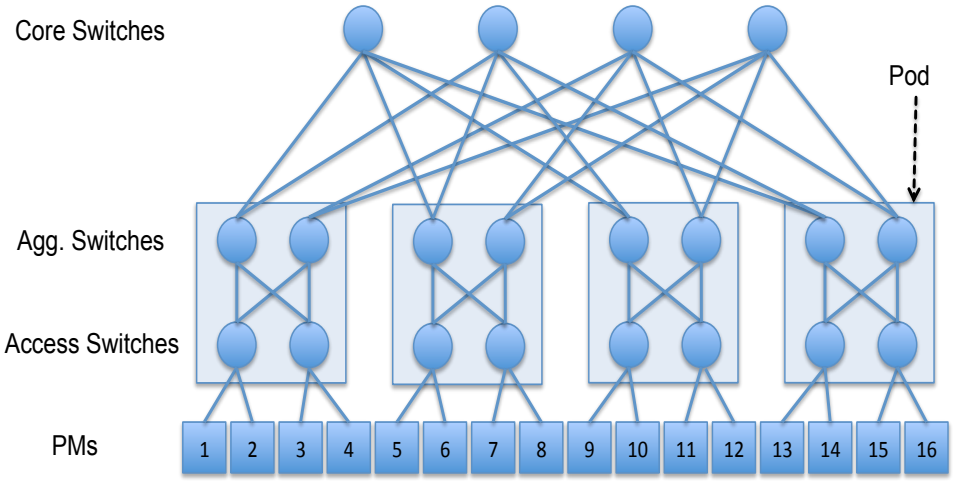

Figure 5.4: Fat-Tree Topology (k=4).

5.2.1 Experimental Setup

In this experiment, all results are simulated using identical VMs types as listed in [71]. In Amazon EC2, the concept of computing units is used to measure the *cpu* capacity for VMs, each compute unit provides an equivalent *cpu* capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor [71]. Without loss of generality, we only specify *cpu* specification dimension for the resource allocation. All physical machines have the same specifications, each physical machine has an 8 core CPU and 64 GB memory. The CPU utilization for each PM is kept under 80%.

The datacenter architecture model used in this experiment is fat-tree, three-tier topology. We simulated the topology using k-port switches to build 128 PMs datacenters (k=8). Using k-port switches in a fat-tree topology gives $\frac{k^2}{2}$ aggregator switches in total; we have k pods and $\frac{k}{2}$ aggregator switches in each pod [65]. Figure 5.4 shows an example of a fat-tree datacenter topology for (k=4). All security middleboxes are placed randomly between switches of different layers.

The SMT solver setup and the machine specifications used to conduct this set of experiments is similar to the experiments explained in §5.1.1.

5.2.2 Results

We evaluate our framework both in terms of the likelihood of finding a satisfiable provisioning and the performance of the provisioning process.

The Effect of Number of Security-enabled Switches on the Probability of Resource Allocation Assignments. Figure. 5.5 shows the probability of successful allocation as the percentage of security enabled switched in the datacenter changes. In this experiment, we assume that the aggregator switches have security services [70]. We use 8-port switches to build a datacenter with 128 PMs. We calculate the probability of a successful allocation for three cases by dedicating: 1, 2, and 4 aggregator switch(es) in each pod to provide security services. The experiment shows

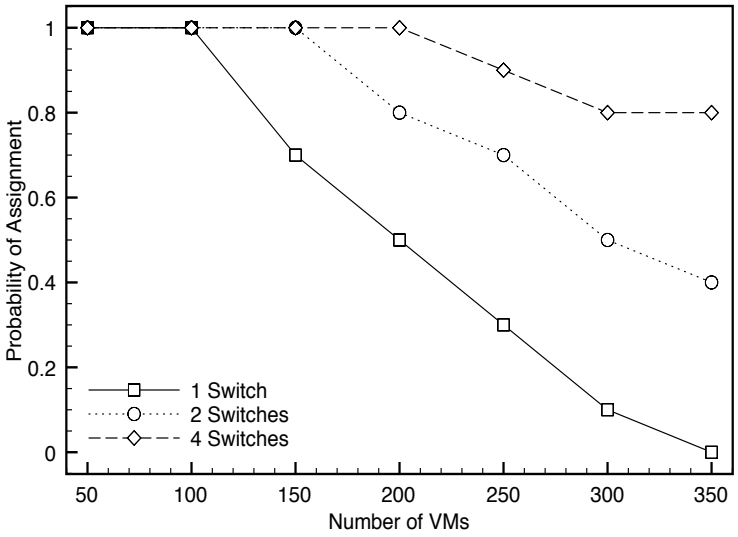

Figure 5.5: The effect of number of security enabled switches on the probability of finding a resource allocation assignment.

that the probability to allocate securely the resources decreases when the percentage of security enabled switched decreases.

Run-Time Overhead. In this set of experiments, we evaluate the run-time overhead to assign security group memberships (i.e., the cloud client perspective) and the run-time overhead to allocate resources (i.e., the cloud provider perspective). We used the following thresholds: g=10, $\mathcal{T}$=25. The distance between VMs is generated randomly in the range [0, 50]. It takes around 0.2 seconds to assign security group memberships for a cloud client with 40 VMs using our grouping model as shown in Figure 5.6. The run-time overhead to assign security groups using EC2 grouping model is slightly less than our grouping model; in our model, more time is needed to find the closest existing group. The run-time overhead for the cloud provider to allocate resources is shown in Figure 5.7. The results show that the time increases exponentially as the number of VMs increases. Yet, it is still within the practical limits to allocate 350 VMs on 128 PMs, the figure shows it takes around 1.5 minutes to allocate such a case.

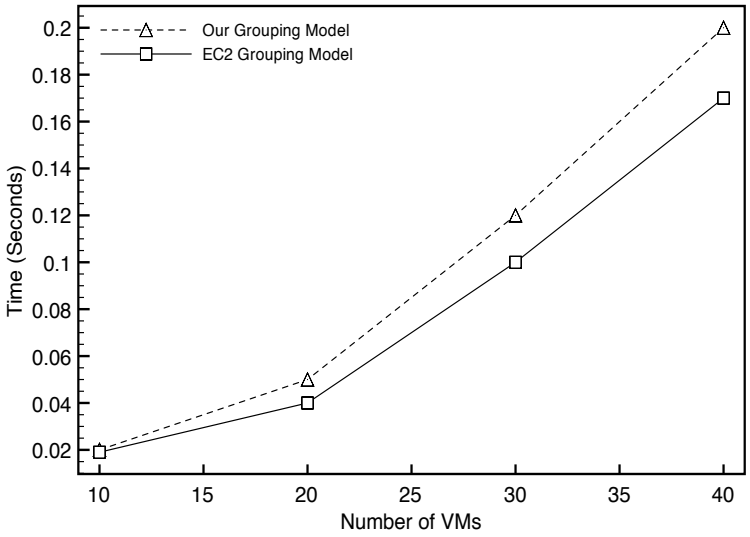

Figure 5.6: Time overhead to assign security group memberships per user.

5.3 Virtual Machine Migration Planning

We have presented a framework for virtual machine migration planning that satisfies temporal relationships associated with bandwidth, collocation, security and workload constraints. Our goal is to find a satisfiable migration plan, i.e., one that satisfies all the constraints and thresholds. The solution includes the migration steps required to assemble resources and make the space needed for the target assignment. We evaluate the framework based on its performance characteristics. We begin by describing the experimental setup and conclude with a summary of results.

5.3.1 Experimental Setup

Without loss of generality, we only specify *cpu* and *memory* specification dimensions for the resources regarding the migration process. The datacenter architecture model used in this experiment is fat-tree, three-tier topology. We simulated the topology using k-port switches to build 16 PMs datacenters (k=4) as shown in Figure 5.4. Using k-port switches in a fat-tree topology gives $\frac{k^2}{2}$ aggregator switches in total; we have k pods and $\frac{k}{2}$ aggregator switches and access switches in each pod, each pod

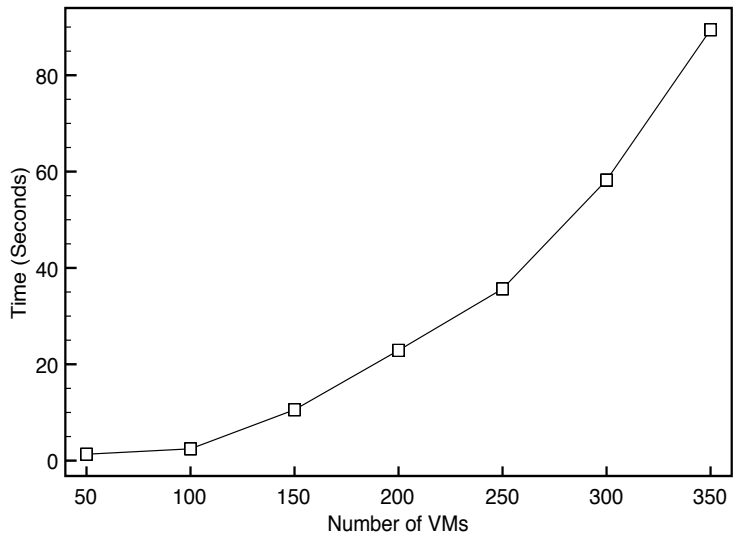

Figure 5.7: Time overhead to allocate resources.

connects $\frac{k^2}{4}$ PMs with $\frac{k^2}{4}$ core switches [65]. All physical machines have the same specifications, each physical machine has an 8 core CPU and 64 GB memory. All VM types used in our simulation are similar to the types used in Amazon EC2 cloud [71].

All evaluation results were simulated on 3.10 GHz quad core CPU with 16 GB memory; the framework uses only one core because it is not a multi-threaded application. For SMT formalization, we used Yices 1.0.36 SMT solver [12] to encode the presented constraints.

The framework is implemented in C++ and all constraints are asserted by invoking Yices C API [12]. All input data and thresholds for the framework are provided through configuration files, which are then parsed and translated into Yices constraints using API functions provided by the solver.

5.3.2 Results

The following set of experiments was conducted to measure the run time overhead to find a migration plan and to measure the percentage of safety violations for the migrated VMs. These two measures depends on several factors: size of migration set, resource utilization percentage, the percentage of critical VMs, the dependency

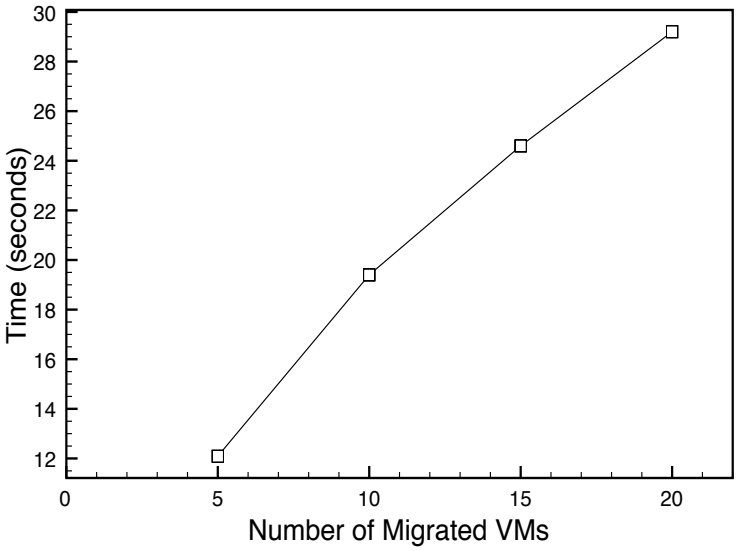

Figure 5.8: Running time overhead to find a migration plan.

threshold (T_D), and the dependency percentage between VMs.

The initial placement of all VMs, π_0, is randomly mapped to the available resources. Then, we randomly select the hotspot PMs that will be switched off. The dependency between VMs is also randomly assigned.

The Impact of Migration Set Size. In this experiment, we evaluate the effect of migration set size on (1) running time overhead to find a satisfiable migration plan, and (2) the fraction of safe migration plans. The fraction of safe migration plans represents the percentage of satisfiable assignments reported by the solver. An unsafe VM migration plan means that some safety requirements are violated. Figure 5.8 depicts the time required to find a VM migration plan. In this experiment, we placed 80 VMs evenly on 16 PMs to achieve 5:1 consolidation ratio; then, we select some PMs to be switched off. We set the percentage of critical VMs to be 30% and the dependency threshold (T_D) to be 2. The linear trend is shown in Figure 5.8, as the number of migrated VMs increases, the running overhead increases linearly. The effect of total number of VMs placed in the datacenter on finding a successful VM migration plan is shown in Figure 5.9. In this experiment, we placed 80, 100, 120

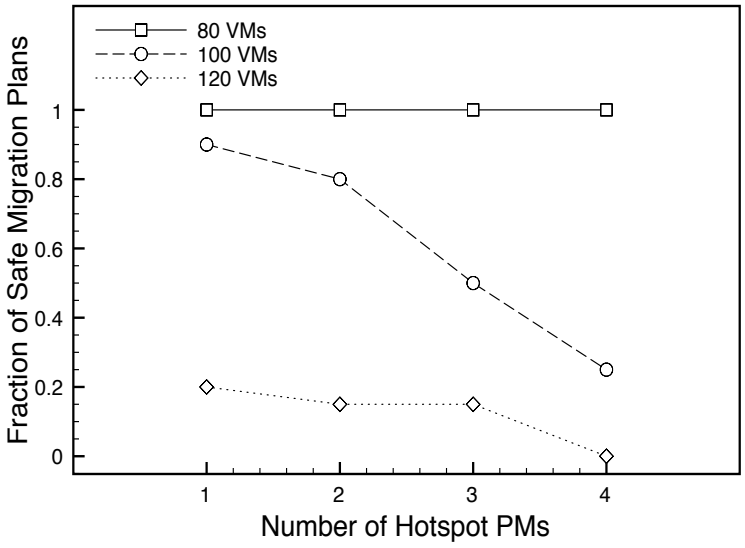

Figure 5.9: The effect of number of placed VMs on fraction of violations.

VMs on the datacenter, respectively. Increasing the number of placed VMs means increasing the consolidation ratio, and means less available resources. As shown in Figure 5.9, placing more VMs in the datacenter affects finding a satisfiable migration plans. The high resource utilization status means that the system cannot accept more VMs to be allocated.

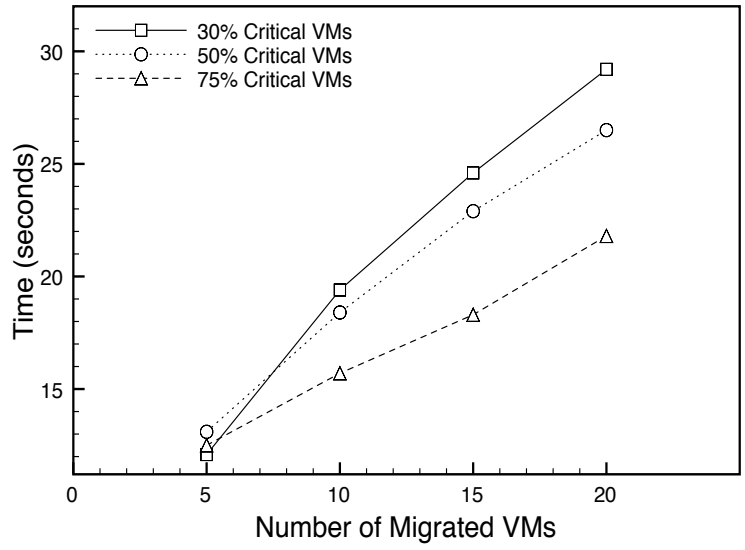

Figure 5.10: The effect of percentage of critical VMs on running time overhead.

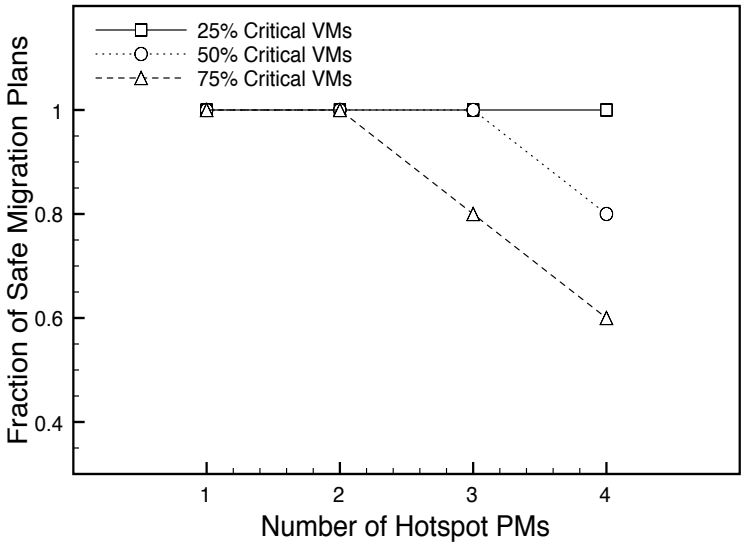

Figure 5.11: The effect of the percentage of critical VMs on fraction of violations.

The Impact of Critical VMs: The effect of the percentage of critical VMs is evaluated in this set of experiments. A critical VM is a VM that cannot be migrated or reallocated, it has to stay on the same PM all the time. Critical VMs limit the search space for the solver and add more restrictions on the migration plan. This, in turn, reduces the time required to find a migration plan. Figure 5.10 shows that increasing the percentage of critical VMs from 30% to 75% reduces the running time overhead from 31 seconds to 22 seconds. The added restrictions by critical VMs decreases the chances to get a migration plan. This restricts the solver to move the VMs around and assemble the needed resources for the migration plan. Figure 5.11 shows that setting 30% of total 80 VMs to be critical VMs does not affect finding satisfiable assignments; while setting the percentage to 75% reduces the percentage of satisfiable migration plans to 60% when switching off 4 PMs (20 VMs to be migrated).

The Impact of Dependency Cost (T_D). Setting $T_D = 2$ means that any two dependent VMs have to be accessible to each other using one access switch at most. While setting $T_D = 4$ means that the dependent VMs are placed in the same pod; otherwise, they can be placed anywhere in the datacenter. In this experiment, we

evaluate the effect of changing the dependency threshold (T_D) on the running time overhead and on the fraction of finding a safe migration plan. Figure 5.12 depicts that there is no extra overhead added as a result of changing the threshold. On the other hand, changing the dependency threshold affects the percentage of finding a migration plan as shown in Figure 5.13. Lowering the dependency threshold (T_D) forces the solver to keep VMs closer to each other during the migration process; in turn, this limits the search space and decreases the chances of getting a safe migration plan.

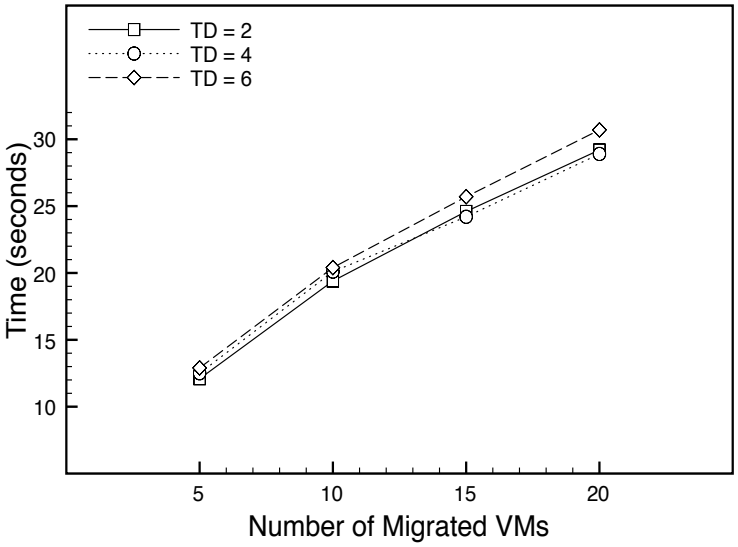

Figure 5.12: The effect of dependency threshold (T_D) on running time overhead.

5.4 Virtual Machine Migration Verification

We have presented a framework for virtual machine migration verification to verify that cloud network configuration is preserved after migration. Our goal is to ensure that the cloud network configuration after VM migration is equivalent to cloud network configuration before migration. Also, we have presented an automated approach to reconfigure firewall policies post VM migration. We evaluate the framework based on its performance characteristics. We begin by describing the experimental setup and conclude with a summary of results.

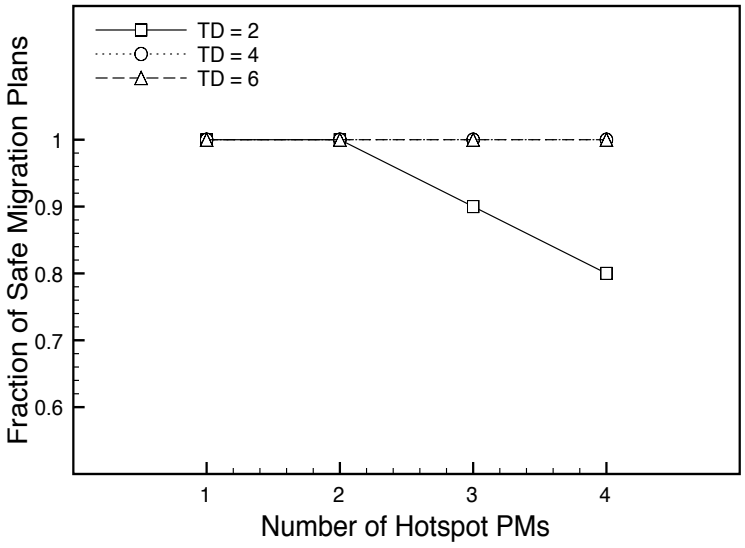

Figure 5.13: The effect of dependency threshold (T_D) on fraction of violations.

5.4.1 Experimental Setup

The experimental setup for this experiment is similar to the experiment performed in section 5.3.1. The framework is implemented in C++ and all firewall policies are encoded in BDDs using BuDDy library [72].

5.4.2 Results

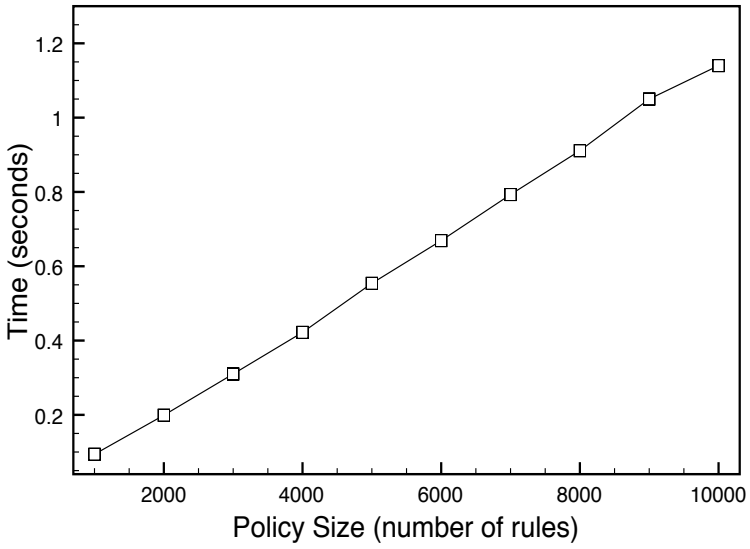

Figure 5.14: Time overhead to build firewall policy BDD.

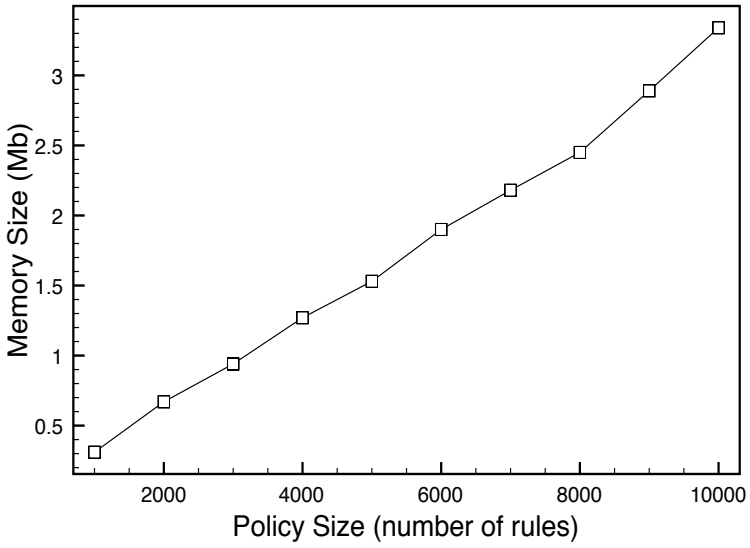

Figure 5.15: Space requirement to build a BDD.

BDD Building Time: Figure 5.14 shows the running time overhead to generate a BDD for a firewall policy. In this experiment, the firewall policies for the cloud network are generated randomly. The policy size (i.e., number of rules) ranges from 1000 to 10000 rules. The figure depicts that the BDD size is linearly dependent on the firewall policy size and it that it takes around 1.14 seconds to build a BDD that represents 10000 rule firewall policy.

BDD Space Requirement: Figure 5.15 shows the space requirement (i.e., BDD size) for generating a BDD. The figure shows that the space requirement for a firewall policy of size 10000 rules is around 3.3 Mb. The BDD size grows linearly as the policy size increases.

VM Migration Verification Overhead: In this experiment, a set of VMs is randomly chosen to be migrated from one host to another host. The migration sequence plan is determined as described in section 4.3. Firewalls are updated according the migration sequence plan, not all firewalls are updated, only the firewalls affected by the VM migration process. Figure 5.16 shows the average time to run all verification constraints. The results show that verification time increases quadratically as the

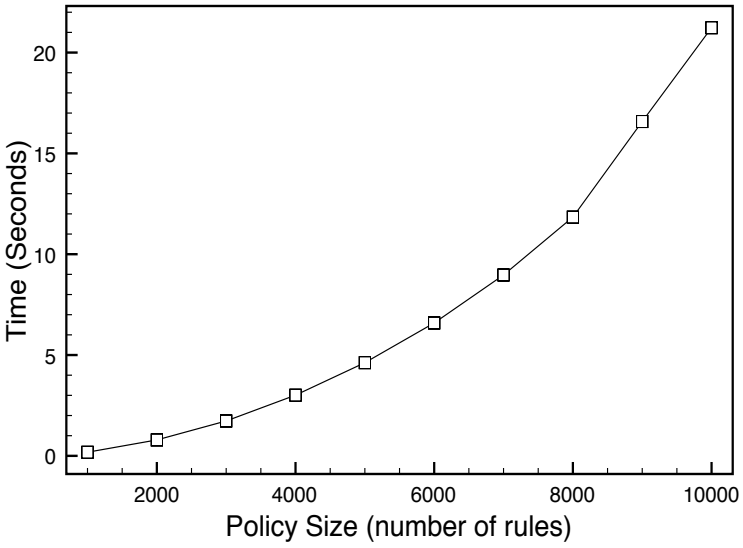

Figure 5.16: Time overhead to run verification constraints.

number of rules increases.

Policy Reconfiguration Overhead: Figure 5.17 shows the average time to generate firewall rules from an existing BDD. Firewall rules are generated by traversing the BDD, every path from the root to a leaf in the BDD will be a firewall rule. The overhead time measured in this experiment includes: 1) the time required to extract/add all network flows related to the migrated VMs from a firewall BDD and 2) the time overhead to traverse all paths in the BDD. The results show that reconfiguration time increases linearly as the number of rules increases.

5.5 Software Defined Networks Misconfigurations

We have presented a framework to mitigate SDN misconfigurations. Our framework can can be used to: (a) verify the consistency of different switches and controllers across federated OpenFlow infrastructures; (b) validate the correctness of the configuration synthesis; (c) debug reachability and security problems; and, (d) assess the consistency of SDN policies. Our framework can also be used as a foundational methodology to conduct "what-if" analysis to study the impact of the new SDN network configurations by simply changing the state in the FlowTables and then

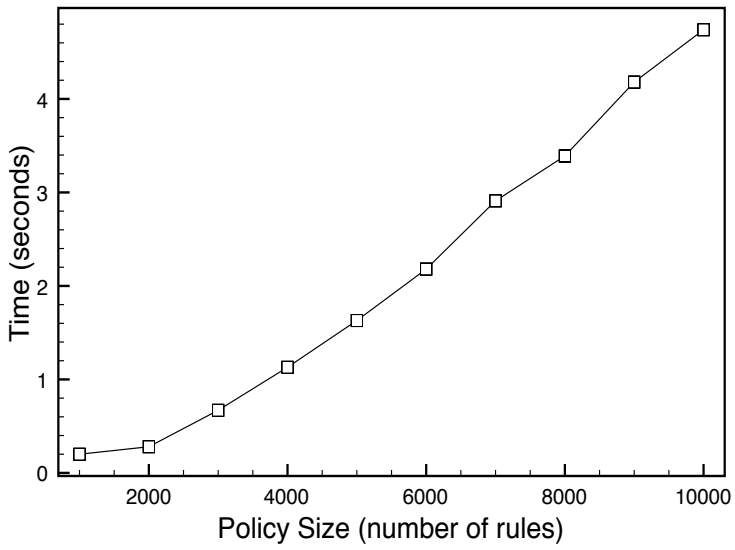

Figure 5.17: Time overhead to reconfigure a firewall policy.

analyzing the effects. In the following, we present an experimental evaluation of our framework. We begin by describing FlowChecker, the simulation environment used to validate our framework. Next, we describe the experimental setup and conclude with a summary of results.

5.5.1 FlowChecker

FlowChecker [8] is an independent centralized server application that receives queries from OpenFlow applications to verify, analyze or debug OpenFlow configuration using our framework. Figure 5.18 shows the architecture for a FlowChecker deployment. Queries can be limited to one domain or extended across federated domains. FlowChecker can run on a master controller (spans number of federated OpenFlow infrastructures). Using FlowChecker, users may write queries to check the validity of certain properties using CTL logic. A validation is done by comparing suggested configuration polices with FlowTables. A report is generated to indicate the conflicts and misconfigurations in the domain.

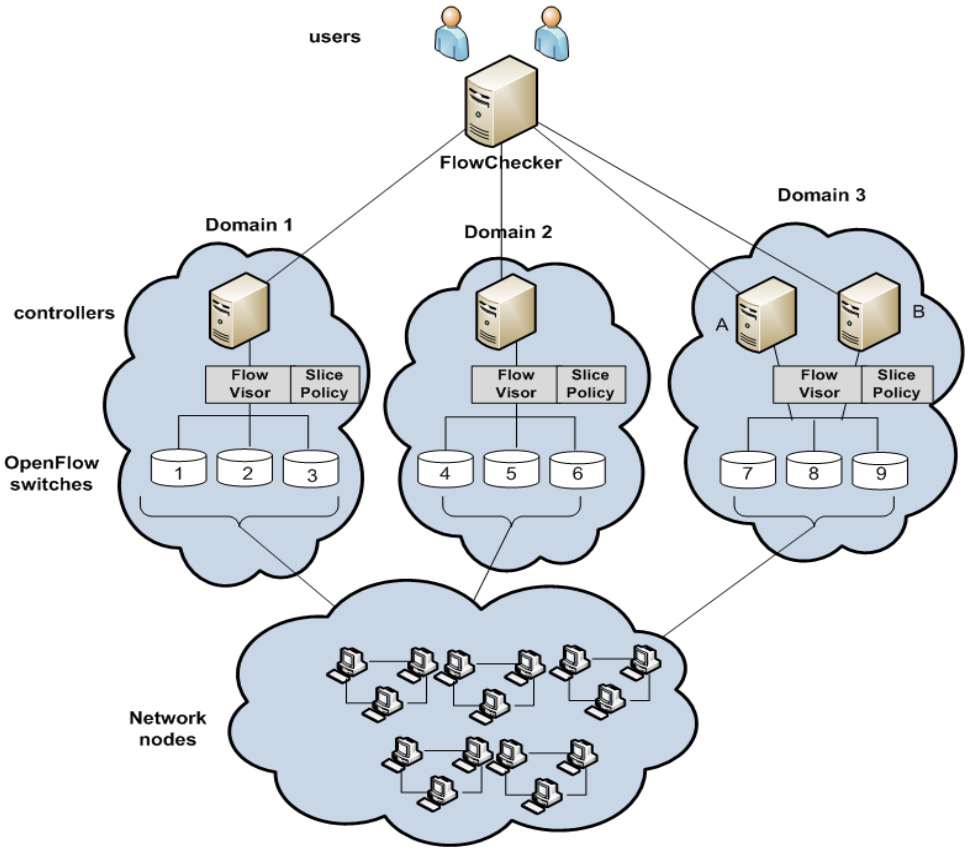

Figure 5.18: FlowChecker connects multiple domains with multiple controllers and OpenFlow switches.

5.5.2 Experimental Setup

In this set of experiments, we examine the performance analysis of our framework as realized in FlowChecker with respect to time and memory space overhead. In order to test many OpenFlow switches policies, a random FlowTable generator was implemented. The FlowTable generator is controlled by many parameters like FlowTable size and the overlapping between FlowTable entries such as redundancy and subset-superset relationship that can appear by using aggregated entries. FlowChecker is implemented using C/C++ language and BuDDy library [72]. BuDDy library provides a rich environment to use BDDs. As we described is section 4.5.3, BDDs are needed to encode FlowTables.

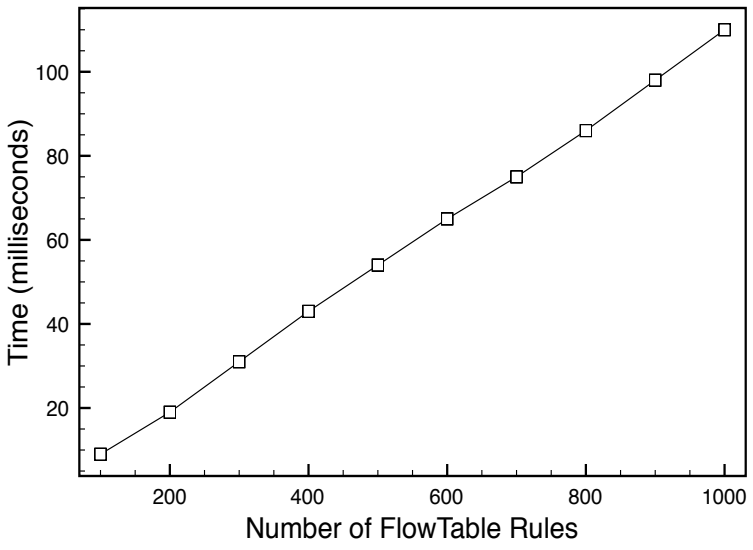

Figure 5.19: Time overhead to build OpenFlow rules BDD.

All evaluation results were simulated on 3.10 GHz quad core CPU with 32 GB memory.

5.5.3 Results

BDD Building Time Overhead: Figure 5.19 shows the running time overhead to generate a BDD for all OpenFlow switches. In this experiment, the FlowTable rules are generated randomly. The policy size (i.e., number of rules) ranges from 100 to 1000 rules. The figure depicts that the BDD size is linearly dependent on the total number of rules and it takes around 110 milliseconds to build a BDD that represents FlowTable 1000 rules.

BDD Space Requirement: Figure 5.20 shows the space requirement (i.e., BDD size) for generating a BDD. The figure shows that the space requirement for 1000 FlowTable rules is around 0.34 Mb. The BDD size grows linearly as the total number of rules increases.

Intra/inter-federated Verification Overhead: We ran many experiments to measure the impact of FlowTable size on the time performance. Figure 5.21 shows

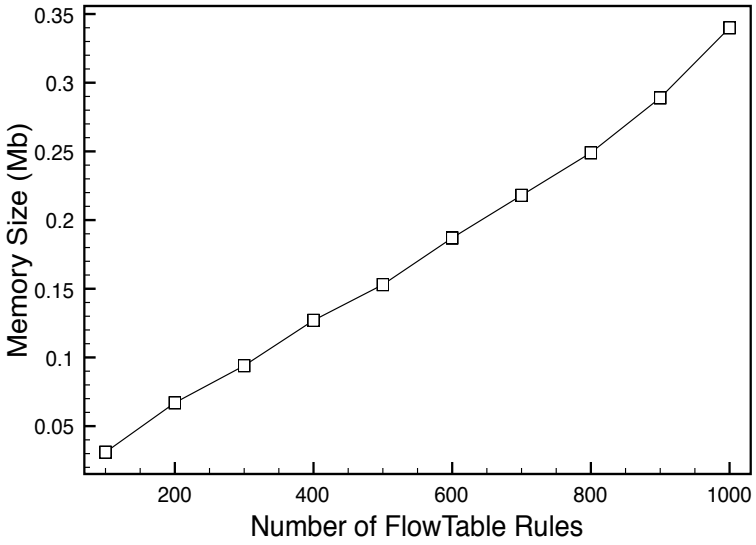

Figure 5.20: Space requirement to build OpenFlow rules BDD.

the time analysis for intra-federated verification for different FlowTable sizes. The quadratic trend appears clearly in Figure 5.21 as the number of OpenFlow switches increase. In the case of intra-federated verification, we need to make pair-wise comparisons for each pair in the domain, as this will give a complexity of $O(n^2)$ to find if there is an inter-switch conflict between n OpenFlow switches.

For inter-federated performance analysis, the quadratic trend will appear as in Figure 5.21 because we still need to do a pairwise comparison for all OpenFlow switches across all domains that we are looking to verify.

Inter-pipeline Verification Overhead: In this experiment, each OpenFlow switch is configured using several FlowTables that are chained together. The aggregate policy of all FlowTables in a switch represents the switch configurations. We injected misconfigured FlowTable rules in several locations. The number of the introduced "set" and "goto table" misconfigured rules is about 10% of the total FlowTable rules. Figure 5.22 shows the average time to run all verification constraints to detect pipeline misconfigurations. The results show that the detection time increases quadratically as the number of misconfigured rules increases.

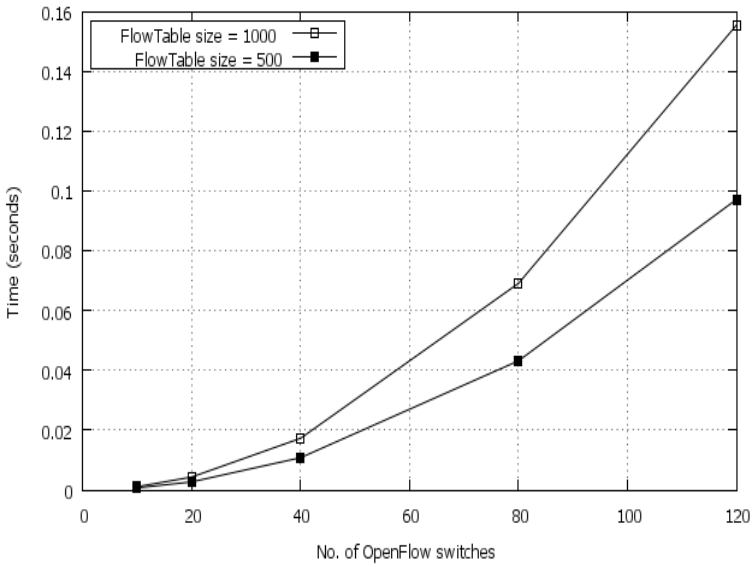

Figure 5.21: FlowChecker time analysis for intra-federated verification.

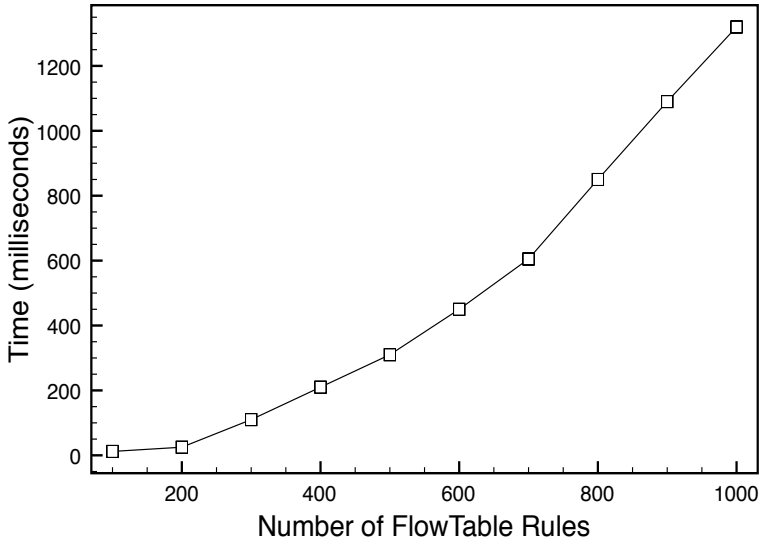

Figure 5.22: Time overhead to detect pipeline misconfigurations in a switch.

CHAPTER 6: CONCLUSION

In this dissertation, we addressed and investigated some challenges of secure and manageable cloud computing infrastructure services. A formal-method based approach composed of five frameworks was presented to address these challenges. We summarize these frameworks as follows.

In §4.1, we addressed the problem of managing VMs ACLs. By utilizing the traffic similarity of VMs, VMs are grouped into groups in which each group is managed by a single ACL. The presented framework used formal-method based approach to encode the distance between VMs using BDDs. The groups are synthesized using a SMT solver. The automatic creation of VMs groups reduces the potential of having policy misconfigurations by reducing the number of ACLs required to manage all VMs by sharing ACLs.

In §4.2, we addressed the problem of secure resource allocation. The presented formal-method based framework considers security requirements at the decision time to allocate resources. The framework reduces the residual risk that result from allocating VMs that have conflicting security requirements on the same host. It also reduces the cost that result from post-security decisions that are needed to solve conflicting requirements.

In §4.3, we addressed the problem of preserving capacity, dependency, and security requirements during VM migration. To meet the challenges of preserving security and performance requirements while VMs are migrated, we presented a formal-method based framework to find the sequence in which VMs need to be migrated in order to preserve security and performance requirements. The migration plan framework models security and performance requirements as constraints satisfaction problem

and relies on SMT solver to find the migration sequence.

In §4.4, we addressed the problem of verifying the equivalence of cloud network configuration before and after VM migration. We presented a formal-method based framework to verify that the updated cloud network configuration is equivalent to the configuration prior to VM migration in terms of its security and network connectivity policies.

Moreover, we addressed the problem of reconfiguring cloud network after VM migration due to the changes of virtual network topologies. Incorrect or inconsistent modifications will introduce policy misconfigurations that affect the security policy and reachability requirements of the virtual network. We presented an incremental, automated, and formal technique to reconfiguration to provide an efficient and configuration-preserving solution to this challenge.

In §4.5, we addressed the SDNs misconfigurations. We argued that cloud datacenters will utilize SDNs to provide better manageability. Before adopting SDNs to host cloud solutions, the risk of SDN misconfigurations needs to be addressed. We presented a formal-method based framework to detect inter/intra policy misconfigurations and FlowTable pipeline misconfigurations in SDNs.

The presented frameworks use formal-method based approaches to provide solutions to the discussed challenges. Across these frameworks, we use BDDs to model policies and ACLs, and constraint satisfaction modeling to encode satisfiability problems.

6.1 Contributions

This dissertation makes the following contributions.

- A formal method-based framework to synthesize access control lists for IaaS virtual machines.

- A formal method-based security-aware resource allocation methodology.

- A formal method-based framework for virtual machine migration planning.
- A formal method-based automated framework to verify configuration consistency before and after virtual machine migration.
- A formal method-based automated framework for post-migration reconfiguration.
- A formal method-based approach to modeling SDN OpenFlow configurations.
- A formal method-based verification interface for SDNs using BDD-based symbolic model checking and temporal logic.
- A formal method-based framework for detecting policy misconfigurations in SDNs.
- A unified approach to combine all previous frameworks under a common methodology.

REFERENCES

[1] "Oracle Fusion Middleware: oracle.com/middleware/index.html."

[2] "IBM WebSphere: www-01.ibm.com/software/websphere/."

[3] G. Menegaz, "The future of cloud is software defined environments: www .ibm.com/developerworks/community/blogs/ibmsyssw/entry/the_future_of_ cloud_is_software_defined_environments?lang=en."

[4] E. Al-Shaer, W. Marrero, and A. El-Atawy, "Network configuration in a box: Towards end-to-end verification of network reachability and security," in *IEEE International Conference of Network Protocols (ICNP'2009)*, October 2009.

[5] "Amazon EC2 Security Groups: docs.amazonwebservices.com/awsec2/latest/ userguide/using-network-security.html."

[6] S. Al-Haj, E. Al-Shaer, and H. Ramasamy, "Security-aware resource allocation in clouds," in *The 10th International Conference on Services Computing (SCC)*, IEEE, 2013.

[7] S. Al-Haj and E. Al-Shaer, "A formal approach for virtual machine migration planning," in *Network and Service Management (CNSM), 2013 9th International Conference on*, pp. 51–58, October 2013.

[8] E. Al-Shaer and S. Al-Haj, "Flowchecker: Configuration analysis and verification of federated openflow infrastructures," in *Proceedings of the 3rd ACM Workshop on Assurable and Usable Security Configuration*, SafeConfig '10, (New York, NY, USA), pp. 37–44, ACM, 2010.

[9] R. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Comput.*, vol. 35, pp. 677–691, August 1986.

[10] H. Andersen, "An introduction to binary decision diagrams," tech. rep., Course Notes on the WWW, 1997.

[11] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach.* Upper Saddle River, NJ, USA: Prentice Hall Press, 3rd ed., 2009.

[12] "Yices SMT solver: yices.csl.sri.com."

[13] M. Balduzzi, J. Zaddach, D. Balzarotti, E. Kirda, and S. Loureiro, "A Security Analysis of Amazon's Elastic Compute Cloud Service," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing, SAC '12*, (New York, NY, USA), pp. 1427–1434, ACM, 2012.

[14] J. Somorovsky, M. Heiderich, M. Jensen, J. Schwenk, N. Gruschka, and L. Lo Iacono, "All your clouds are belong to us: security analysis of cloud management interfaces," in *Proceedings of the 3rd ACM CCSW*, (New York, NY, USA), pp. 3–14, ACM, 2011.

[15] L. Sumter, "Cloud computing: Security risk," in *Proceedings of the 48th Annual Southeast Regional Conference, ACM SE '10*, (New York, NY, USA), pp. 112:1–112:4, ACM, 2010.

[16] J. Wei, X. Zhang, G. Ammons, V. Bala, and P. Ning, "Managing security of virtual machine images in a cloud environment," in *Proceedings of the 2009 ACM CCSW*, (New York, NY, USA), pp. 91–96, ACM, 2009.

[17] X. Meng, C. Isci, J. Kephart, L. Zhang, E. Bouillet, and D. Pendarakis, "Efficient resource provisioning in compute clouds via vm multiplexing," ICAC '10, (New York, NY, USA), pp. 11–20, ACM, 2010.

[18] M. Alicherry and T. Lakshman, "Network aware resource allocation in distributed clouds," in *INFOCOM, 2012 Proceedings IEEE*, pp. 963 –971, March 2012.

[19] G. Lee, N. Tolia, P. Ranganathan, and R. Katz, "Topology-aware resource allocation for data-intensive workloads," *SIGCOMM Comput. Commun. Rev.*, vol. 41, pp. 120–124, January 2011.

[20] H. Goudarzi and M. Pedram, "Multi-dimensional SLA-based resource allocation for multi-tier cloud computing systems," in *Proceedings of the 2011 IEEE 4th International Conference on Cloud Computing, CLOUD*, (Washington, DC, USA), pp. 324–331, IEEE Computer Society, 2011.

[21] N. Calcavecchia, O. Biran, E. Hadad, and Y. Moatti, "VM placement strategies for cloud scenarios," in *IEEE 5th International Conference on Cloud Computing (CLOUD)*, pp. 852 –859, June 2012.

[22] G. Wei, A. Vasilakos, Y. Zheng, and N. Xiong, "A game-theoretic method of fair resource allocation for cloud computing services," *J. Supercomput.*, vol. 54, pp. 252–269, November 2010.

[23] S. Srikantaiah, A. Kansal, and F. Zhao, "Energy aware consolidation for cloud computing," in *Proceedings of the 2008 conference on Power aware computing and systems*, HotPower'08, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2008.

[24] D. Ardagna, B. Panicucci, M. Trubian, and L. Zhang, "Energy-aware autonomic resource allocation in multitier virtualized environments," *IEEE Transactions on Services Computing*, vol. 5, pp. 2 –19, Jan-Mar 2012.

[25] J. Piao and J. Yan, "A network-aware virtual machine placement and migration approach in cloud computing," in *Grid and Cooperative Computing (GCC), 2010 9th International Conference on*, pp. 87–92, 2010.

[26] V. Shrivastava, P. Zerfos, K.-W. Lee, H. Jamjoom, Y.-H. Liu, and S. Banerjee, "Application-aware virtual machine migration in data centers," in *INFOCOM, 2011 Proceedings IEEE*, pp. 66–70, 2011.

[27] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif, "Black-box and gray-box strategies for virtual machine migration," in *Proceedings of the 4th USENIX conference on Networked systems design & implementation*, NSDI'07, (Berkeley, CA, USA), pp. 17–17, USENIX Association, 2007.

[28] Y. Ma, H. Wang, J. Dong, Y. Li, and S. Cheng, "Me2: Efficient live migration of virtual machine with memory exploration and encoding," in *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, pp. 610–613, 2012.

[29] M. Al Shayeji and M. Samrajesh, "An energy-aware virtual machine migration algorithm," in *Advances in Computing and Communications (ICACC), 2012 International Conference on*, pp. 242–246, 2012.

[30] M. Mishra, A. Das, P. Kulkarni, and A. Sahoo, "Dynamic resource management using virtual machine migrations," *Communications Magazine, IEEE*, vol. 50, no. 9, pp. 34–40, 2012.

[31] X. Zhang, Z.-Y. Shae, S. Zheng, and H. Jamjoom, "Virtual machine migration in an over-committed cloud," in *Network Operations and Management Symposium (NOMS), 2012 IEEE*, pp. 196–203, 2012.

[32] W. Voorsluys, J. Broberg, S. Venugopal, and R. Buyya, "Cost of virtual machine live migration in clouds: A performance evaluation," in *Proceedings of the 1st International Conference on Cloud Computing*, CloudCom '09, (Berlin, Heidelberg), pp. 254–265, Springer-Verlag, 2009.

[33] S. Ghorbani and M. Caesar, "Walk the line: consistent network updates with bandwidth guarantees," in *Proceedings of the first workshop on Hot topics in software defined networks*, HotSDN '12, (New York, NY, USA), pp. 67–72, ACM, 2012.

[34] X. Li, Q. He, J. Chen, K. Ye, and T. Yin, "Informed live migration strategies of virtual machines for cluster load balancing," in *Proceedings of the 8th IFIP international conference on Network and parallel computing*, NPC'11, (Berlin, Heidelberg), pp. 111–122, Springer-Verlag, 2011.

[35] E. Al-Shaer and H. Hamed, "Discovery of policy anomalies in distributed firewalls," in *In IEEE INFOCOM '04*, pp. 2605–2616, 2004.

[36] H. Hamed, E. Al-Shaer, and W. Marrero, "Modeling and verification of ipsec and vpn security policies," in *ICNP '05: Proceedings of the 13TH IEEE International Conference on Network Protocols*, pp. 259–278, 2005.

[37] M. Gouda and X. Liu, "Firewall design: Consistency, completeness, and compactness," in *The 24th IEEE Int. Conference on Distributed Computing Systems (ICDCS'04)*, March 2004.

[38] M. Gouda, A. Liu, and M. Jafry, "Verification of distributed firewalls," in *Global Telecommunications Conference, 2008. IEEE GLOBECOM 2008. IEEE*, pp. 1–5, November 2008.

[39] L. Yuan, J. Mai, Z. Su, H. Chen, C. Chuah, and P. Mohapatra, "FIREMAN: A toolkit for firewall modeling and analysis," in *IEEE Symposium on Security and Privacy (SSP'06)*, May 2006.

[40] A. Gawanmeh and S. Tahar, "Modeling and verification of firewall configurations using domain restriction method," in *Internet Technology and Secured Transactions (ICITST), 2011 International Conference for*, pp. 642–647, December 2011.

[41] H. Acharya and M. Gouda, "Firewall verification and redundancy checking are equivalent," in *INFOCOM, 2011 Proceedings IEEE*, pp. 2123–2128, April 2011.

[42] S. Al-Haj, P. Bera, and E. Al-Shaer, "Build and test your own network configuration," in *Security and Privacy in Communication Networks*, vol. 96 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pp. 522–532, Springer Berlin Heidelberg, 2012.

[43] Y. Jarraya, A. Eghtesadi, M. Debbabi, Y. Zhang, and M. Pourzandi, "Cloud calculus: Security verification in elastic cloud computing platform," in *Collaboration Technologies and Systems (CTS), 2012 International Conference on*, pp. 447–454, May 2012.

[44] Y. Jarraya, A. Eghtesadi, S. Sadri, M. Debbabi, and M. Pourzandi, "Verification of firewall reconfiguration for virtual machines migrations in the cloud," *Computer Networks*, vol. 93, Part 3, pp. 480 – 491, 2015. Cloud Networking and Communications {II}.

[45] A. Eghtesadi, Y. Jarraya, M. Debbabi, and M. Pourzandi, "Preservation of security configurations in the cloud," in *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, pp. 17–26, March 2014.

[46] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, 2008.

[47] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Flowvisor: A network virtualization layer," Tech. Rep. OpenFlow Technical Report 2009-1, Deutsche Telekom Inc. R&D Lab, Stanford University, Nicira Networks, October 2009.

[48] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "Nox: towards an operating system for networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 3, pp. 105–110, 2008.

[49] S. Son, S. Shin, V. Yegneswaran, P. Porras, and G. Gu, "Model checking invariant security properties in openflow," in *2013 IEEE International Conference on Communications (ICC)*, pp. 1974–1979, June 2013.

[50] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, (New York, NY, USA), pp. 49–54, ACM, 2012.

[51] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford, "A nice way to test openflow applications," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2012.

[52] H. Hu, W. Han, G.-J. Ahn, and Z. Zhao, "Flowguard: Building robust firewalls for software-defined networks," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, (New York, NY, USA), pp. 97–102, ACM, 2014.

[53] N. Feamster and H. Balakrishnan, "Detecting BGP configuration faults with static analysis," in *NSDI*, 2005.

[54] T. Griffin and G. Wilfong, "On the correctness of IBGP configuration," in *SIGCOMM '02: Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, (New York, NY, USA), pp. 17–29, ACM, 2002.

[55] R. Alimi, Y. Wang, and Y. Yang, "Shadow configuration as a network management primitive," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 4, pp. 111–122, 2008.

[56] R. Mahajan, D. Wetherall, and T. Anderson, "Understanding BGP misconfiguration," in *SIGCOMM '02: Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, (New York, NY, USA), pp. 3–16, ACM, 2002.

[57] R. Bush and T. Griffin, "Integrity for virtual private routed networks," in *IEEE INFOCOM 2003*, vol. 2, pp. 1467– 1476, 2003.

[58] G. Xie, J. Zhan, D. Maltz, H. Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford, "On static reachability analysis of ip networks," in *IEEE INFOCOM 2005*, vol. 3, pp. 2170– 2183, 2005.

[59] ONF, "SDN security considerations in the data center," October 2013. `https://www.opennetworking.org/solution-brief-sdn-security-considerations-in-the-data-center`.

[60] "Amazon EC2: aws.amazon.com/ec2."

[61] O. Santos, *End-to-end Network Security: Defense-in-depth.* Cisco Press, first ed., 2007.

[62] "Common Vulnerability Scoring System (CVSS): www .first.org/cvss."

[63] M. Ahmed, E. Al-Shaer, M. Taibah, and L. Khan, "Objective risk evaluation for automated security management," *J. Netw. Syst. Manage.*, vol. 19, pp. 343–366, September 2011.

[64] "IBM SmartCloud Enterprise server configurations: www-935.ibm.com/services/bn/en/cloud-enterprise/tab-details-server-configurations.html."

[65] X. Meng, V. Pappas, and L. Zhang, "Improving the scalability of data center networks with traffic-aware virtual machine placement," INFOCOM'10, (Piscataway, NJ, USA), pp. 1154–1162, IEEE Press, 2010.

[66] Z. Tavakoli, S. Meier, and A. Vensmer, "A framework for security context migration in a firewall secured virtual machine environment," in *Information and Communication Technologies* (R. Szabó and A. Vidács, eds.), vol. 7479 of *Lecture Notes in Computer Science*, pp. 41–51, Springer Berlin Heidelberg, 2012.

[67] Y. Zhang, A. Juels, M. Reiter, and T. Ristenpart, "Cross-vm side channels and their use to extract private keys," in *Proceedings of the 2012 ACM CCS*, (New York, NY, USA), pp. 305–316, ACM, 2012.

[68] E. Al-Shaer and H. Hamed, "Firewall policy advisor for anomaly detection and rule editing," in *IEEE/IFIP Integrated Management (IM'2003)*, March 2003.

[69] E. Al-Shaer, H. Hamed, R. Boutaba, and M. Hasan, "Conflict classification and analysis of distributed firewall policies," *IEEE Journal on Selected Areas in Communications (JSAC)*, vol. 23, October 2005.

[70] "Cisco data center infrastructure 2.5 design guide."

[71] "Amazon EC2 instance types: aws.amazon.com/ec2/instance-types."

[72] J. Lind-Nielsen, "The BuDDy OBDD package." http://www.bdd-portal.org/buddy.html.