RZ 3859 (# ZUR1308-055) 12/20/2013
Computer Science 15 pages

Research Report

Modeling and Analysis of Dynamic Infrastructure Clouds

Sören Bleikertz

IBM Research – Zurich, 8803 Rüschlikon, Switzerland
Email: sbl@zurich.ibm.com

Thomas Groß

University of Newcastle Upon Tyne
Email: thomas.gross@newcastle.ac.uk

Sebastian Mödersheim

DTU Informatics
Email: samu@imm.dtu.dk

Modeling and Analysis of Dynamic Infrastructure Clouds

Sören Bleikertz
IBM Research - Zurich
sbl@zurich.ibm.com

Thomas Groß
University of Newcastle upon Tyne
thomas.gross@newcastle.ac.uk

Sebastian Mödersheim
DTU Informatics
samo@imm.dtu.dk

***Abstract*—Misconfigurations and insider attacks contribute to one of the major technical risk in multi-tenant cloud computing: the lack of resource isolation. Breaches in tenant isolation put both the cloud provider as well as the consumers at risk. The dynamic nature of infrastructure clouds increases the risk for misconfigurations due to their self-service administration and rapid provisioning.**

We tackle this challenge by establishing a practical security system that proactively analyzes changes induced by cloud management operations with regard to a security policy. We achieve this by contributing the first formal model of cloud management operations and their impact on a virtualized infrastructure. Our approach combines such a model of operations with a security policy verification as well as an information flow analysis suited for isolation policies. Our system finds practical applications in change planning as well as in auditing of changes at run-time. We evaluate our system for virtualized infrastructures in laboratory and production settings, and it yields a performance suitable for applications in practice.

I. Introduction

Multi-tenant infrastructure clouds offer cloud consumers self-service access to a shared physical infrastructure, be it in computing, storage or networking. While administrators of the cloud provider govern the infrastructure as a whole and the tenant administrators operate in partitioned logical resource pools, both groups change the configuration of the infrastructure. For example, they create new machines, modify or delete existing ones, causing large numbers of machines to appear and disappear, which leads to the phenomenon of server sprawl [11]. Therefore, self-service administration, dynamic provisioning and elastic scaling lead to a great number of configuration changes and a dynamic system. The complexity of the system as a whole emerges from these configuration changes and dynamic behavior.

Misconfigurations and insider attacks are the adverse results of such dynamic and complex systems. Indeed, even if committed unintentionally, misconfigurations are among the most prominent causes for security failures in IT infrastructure [15]. Notably, the ENISA report on cloud security risks [10] names isolation failure as major technical risk, with misconfiguration as the notable root vulnerability. Consider an administrator of a cloud provider, who unintentionally commits a configuration change that breaks the isolation among tenants in the infrastructure. This puts both the cloud provider as well as the consumers at great risk due to potential loss of reputation and the breach of confidential data. The CSA threat report [7] and the ENISA report agree upon insider attacks as a TOP 10 cloud security risk as well as malicious insiders as a "very high impact" risk [10].

In combating misconfigurations and insider attacks, the assessment of configuration changes and rigorous enforcement of security policies is a crucial requirement. We tackle this challenge with a model-based approach for assessing configuration changes and their impact on the security compliance of a virtualized infrastructure. Our model is based on a graph representation of the topology and configuration of a virtualized infrastructure, and we contribute the first formal model of management operations, the *operations transition model*, that captures how such operations change the infrastructure's topology and configuration. We express the operations as transformations of a graph model of the infrastructure, which is based upon the formalism of graph transformation [17]. Furthermore, we integrate the specification of security policies as well as an information flow analysis suited for isolation policies. Based on our model, we design and implement a practical security system, called our system, to assess and eventually proactively mitigate misconfigurations. Concretely, we make the following contributions:

1) We are the first to propose an operations model, a model of configuration and topological changes in virtualized infrastructures. Overall, we propose a unified model that combines topological and configuration changes, information flow analysis, as well as security policies, in order to enable the security assessment of dynamically changing virtualized infrastructures. **2)** We implemented a prototype that targets VMware virtualized infrastructures and leverages *GROOVE* as a graph transformation system. We evaluate the system in a laboratory and production environments and obtain reasonable performance. **3)** We explore two practical application scenarios for our system: i) change management, where the desired changes are specified by an administrator and submitted for assessment; and ii) run-time auditing of configuration changes, in which changes are assessed and security violations accounted for while being performed by an administrator. We prepare the ground for a run-time mitigation of misconfigurations and enforcement of security policies.

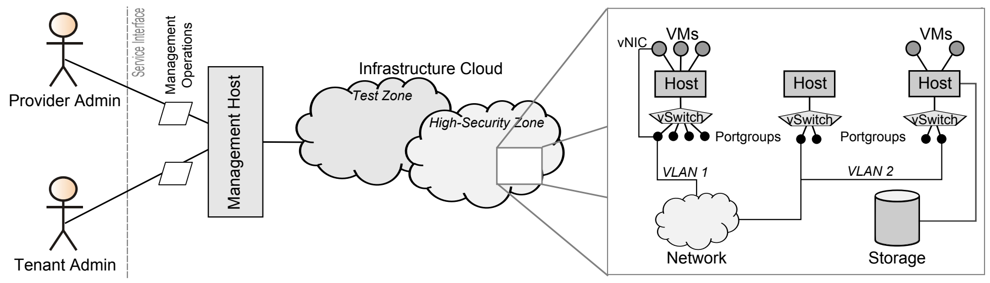


Figure 1: Overview of the System Model.

II. System and Security Model

A. System Model

An infrastructure cloud, also named Infrastructure as a Service (IaaS), is a cloud model offering for computing, networking, and storage resources. Typically, such resources are provided in a virtualized form, e.g., as a virtual machine (VM) computing. The main ingredients for an infrastructure cloud are the virtualized infrastructure, which provides the virtual resources, and a management host, which exposes a management interface to an administrator to control the infrastructure. The infrastructure is only managed through the well-defined management interface. Management operations allow the administrators to re-configure parameters of components as well as to change the topology, e.g., by creating or migrating a VM.

In Fig. 1 we illustrate our model of an infrastructure cloud. We differentiate between provider and tenant administrators, where the provider administrators govern the entire virtualized infrastructure, and tenant administrators manage an assigned logical resource pool. Provider administrators are much more privileged and we focus our analysis on their operations and changes, as they may cause significant isolation failures. Furthermore, we differentiate between a high-level interface, which is usually exposed to cloud consumers by public clouds such as Amazon EC2, and a low-level interface, which is exposed by virtualization systems in private clouds such as VMware. Whereas the high-level interface provides operations such as creating a virtual machine, the low-level interface allows the administrator to change all parts of the virtualized infrastructure. The latter *service interface* bears much more complexity and is prone to induce misconfigurations, and therefore is the focus of our analysis.

In our system model, we consider *security zones* as a logical grouping of virtual resources, which should be isolated from each other. For example, in our illustration we have a test security zone as well as high-security zone, but one could also think about one zone per tenant. A close-up of one of these zones shows the topology of the virtualized infrastructure. In there, hosts provide networking to VMs by virtual switches that connects the VMs to the network. A virtual switch contains virtual ports, to which the VMs are connected via a virtual network card (vNIC). Virtual ports are aggregated into *port groups*, which apply a common configuration to a group of virtual ports. Virtual LANs (VLANs) allow a logical separation of network traffic between VMs of different security zones by assigning distinct VLAN IDs to the associated port groups. Storage resources are also connected to the hosts and exposed to the VMs, but they are not part of our focus in this paper.

B. Graph Representation of the System

We represent the topology and configuration of the virtualized infrastructure in a graph representation, for which the notion of a *Realization* model has been proposed by Bleikertz et al. [4]. The vertices of such a graph represents the components of the virtualized infrastructure, e.g., physical servers or virtual machines, and the edges model the relationship among the elements, thereby capturing the topology of the system. Furthermore, nodes are typed, e.g., vmachine, and attributed to capture further properties and configuration aspects of each element. We consider the model as a directed, node and edge typed, and attributed graph. We distinguish different edge types, where the edges of the foundational *Realization* model are labeled with type real in accordance to the name of the model.

In order to build the model we require sufficient information on the topology and configuration of a virtualized infrastructure. An existing method for constructing a model [4] consists of two steps: *Discovery* and *Translation*. The discovery extracts the configuration from the hypervisors or management system of heterogeneous virtualized infrastructures, and translates the extracted configuration data into the model.

C. Threat Model

We draw upon the dependability taxonomy [1] to establish a well-defined threat model. We consider that even if administrators are honest, they can still make mistakes that lead to a security breach. Therefore, as core threat, we model non-malicious human-made faults, which may be deliberate and non-deliberate, accidental or due to incompetence. This covers the classes *intent*, where deliberate faults are a result of a harmful decision and non-deliberate faults are introduced without awareness, as well as *capability*, where accidental faults are introduced inadvertently and incompetence faults result from a lack of professional competence. Consider the following examples of possible attack scenarios.

- An administrator produces a typo when entering a new port group's VLAN ID (non-malicious, non-deliberate, accidental). If the ID collides with one that is already used in another security zone, we have an isolation breach between zones.
- An administrator intentionally sets the VLAN IDs of port groups of two different zones to the same value, to save precious VLAN IDs, believing that the networks are separated by means of physical isolation or higher level network isolation, e.g., based on firewalls (non-malicious, deliberate, incompetence).

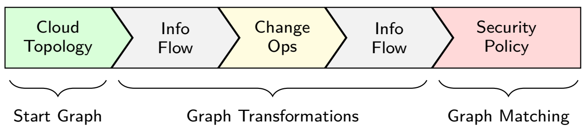
The administrator behavior comes with some fairness constraints: Provider and tenant administrators will attempt to issue commands in a well-defined way through the service interface. Administrators will take into account feedback from a security analysis or an audit. As part of the system foundations, we require that the extraction of the infrastructure configuration provides an authentic view of the configuration.

III. A Model of Dynamic Virtualized Infrastructures

The model is the foundation for the proactive analysis of changes in virtualized infrastructures. We capture multiple aspects relevant for the analysis and integrate them into a unified model. For that, we need to represent the topology and configuration of the virtualized infrastructure, establish how the infrastructure can be changed by cloud management operations, and verify the security policy. As we are focusing on isolation properties, we further need to determine information flows in the system. All aspects in this model need to cope with the dynamic behavior of virtualized infrastructures.

Fig. 2 illustrates how the different parts of the model are intertwined and indicates how the model will be employed in the analysis. The model builds on a graph representation as well as transformations of graphs to capture the dynamic behavior. The graph representation of the cloud topology and configuration (cf. §II-B) forms the *Start Graph*, which will be the starting point of the subsequent analysis. The information flow analysis as well as the model of cloud management operations perform *Graph Transformations* on the start graph: They are changing parts of a given graph and output a modified one. Finally, we match the security policy against each transformed graph to find violations (*Graph Matching*).

Figure 2: Overview of Model Composition.



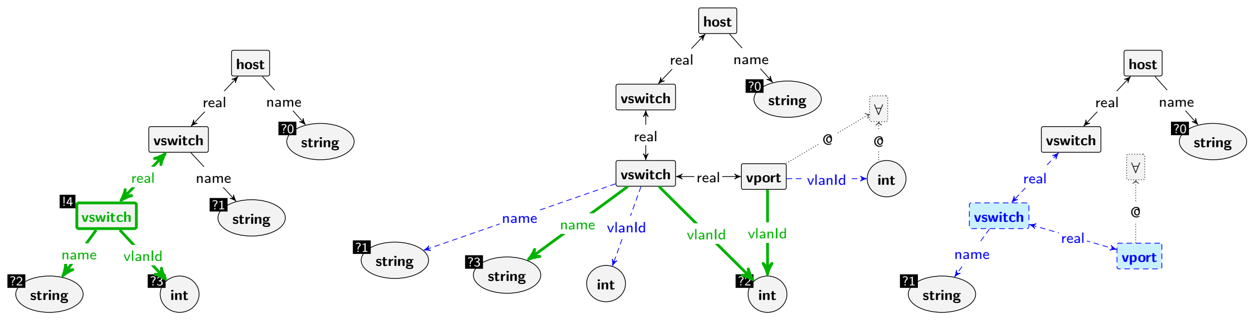
Graph Transformations: Graph transformations are essential for our model. In order to better understand them, we will briefly introduce their foundational concepts (For details on the theory we refer to Rozenberg [17]): We have a graph transformation rule p, also called a *production*, in the form of $p : L \xrightarrow{r} R$, where graphs L and R are denoted the left hand side (LHS) and right hand side (RHS), respectively. The *production morphism* r establishes a partial correspondence between elements in the LHS and the RHS of a production, which determines the nodes and edges that have to be preserved, deleted, or created. A *match* m finds an occurrence of L in a given graph G, then $G \overset{p,m}{\Longrightarrow} H$ is an application of a production p, where H is a derived graph. H is obtained by replacing the occurrence of L in G with R.

Many graph transformation tools have been developed in the past, among them are: *GROOVE*, AGG, GRGen, and PROGRES. For this work, we decided to use *GROOVE* [12] as our graph transformation environment. It is a general-purpose graph transformation tool that enables an expressive specification of production rules, e.g., by providing quantifications and path constructions using regular expressions. Furthermore, its imperative control language allows us to control the application of rules as well as to use parametrized rules, in which parameters are passed from a control program to a production rule.

A. Modeling of Infrastructure Changes

The *Operations Transition Model* captures how management operations change topology and configuration of a virtualized infrastructure. Our As we aim to create a practical security system for virtualized infrastructures, we focus our modeling efforts on VMware and its operations. We model each operation as a graph production rule, which takes the graph representation of the virtualized infrastructure as input and transforms it into a modified one.

1) Subset of Operations: Whereas VMware API (v5.0) consists of 545 methods [19], only a subset of them is relevant for our operations transition model. Many operations do not affect the topology or configuration of the virtualized infrastructure. However, they deal with VMware-specific management and operations aspects such as licensing and patch management, handling of administrative sessions, or diagnostics and alarms. For now, we model a security-relevant subset of VMware management operations. We consider changes to the



(a) AddPortGroup (b) UpdatePortGroup (c) RemovePortGroup

Figure 3: PortGroup Operations in *GROOVE*.

network configuration, in particular the configuration of port groups, virtual switches, and virtual network interfaces. We model methods that create, modify, and delete such network elements. Whereas a port group's network policy contains interesting properties, such as whether a NIC is in promiscious mode, we omit it in favor of a clear focus on the isolation properties of the topology. Furthermore, we consider the creation of virtual machines. We leave extending this subset of modeled operations, e.g., including storage, to future work: IT is a time-consuming but otherwise neither challenging nor enlightening task. However, this subset already enables the analysis of interesting topology and configuration changes in the areas of virtual networking and compute resources, and, thereby, the verification of isolation breaches.

2) Modeling Approach: For any existing real-world virtualized infrastructure like VMware, the API documentation does not offer a precise formal definition and model, but rather a semi-formal description of the operations, parameters as well as the preconditions and effects that the operations have. A contribution of this paper is to create a formal model that allows for precise statements to be made and proved or refuted. It is of course not possible to formally prove the correctness of such a model itself, however there is a methodology to obtain a "good" model by combining two directions.

The first direction is to follow the documentation and translate the documented effects into our graph model. The second is to experiment with the real implementation, to evaluate empirically whether the actual operations indeed affect the infrastructure as our model predicts. To study these experiments we need to translate the infrastructure topology into a graph *before* and *after* the operation has been performed, and check whether the resulting graph transformation matches our model of the operation.

3) Modeling Examples: Fig. 3 shows examples of the creation, updating, and deletion of the virtual networking of VMware. The so called port groups have a direct impact on the network isolation of tenants as they determine the VLAN configuration. We adopted the visual notation of production rules from *GROOVE* to specify the rules of this paper. This notation makes the key points of our paper easy to grasp, while it has a well-defined formal semantics. Each graph production rule is itself described by a graph where nodes and edges can have one of the following colors/styles:

- **Readers** (black): nodes and edges that need to be matched in the graph for the rule to be applicable; they are preserved in the transformation. For example, the host node in Fig. 3a.
- **Creators** (bold green): newly added nodes and edges. The lower vswitch and corresponding edges in Fig. 3a are creators.
- **Erasers** (thin dashed blue): like the readers, but will be deleted by the transformation. For instance, the lower vswitch in Fig. 3c.
- **Embargoes** (bold dashed red): negative application conditions (NAC); the rule cannot be applied if any matching nodes and edges are present.

By combining *Embargoes* and *Creators*, we can express a *conditional new*, where nodes and/or edges are only created if they do not exist yet. For example, we make use of conditional news in the information flow rules (Fig. 4, §III-B1). The bold label of nodes represent their type, e.g., a host.

Our example operations and their models are a representation for three common classes of operations that i) create new infrastructure elements and therefore new nodes and edges in the graph model, ii) change infrastructure elements' attributes and the topology, which translates to the addition and removal of edges to graph nodes and to data nodes that hold the attributes' values, iii) delete infrastructure elements and thereby also nodes and edges in the model. Parametrized production rules are essential as management operations operate on a specific part of the virtualized infrastructure, e.g., on a specific host given by a hostname parameter, and set or change attributes based on given parameters.

Table I: VMware PortGroup Operations [19]

AddPortGroup:	
__this	host network system reference
portgrp	HostPortGroupSpec specified below:
UpdatePortGroup:	
__this	host network system reference
pgName	port group name
portgrp	HostPortGroupSpec
HostPortGroupSpec:	
name	name of the port group
policy	network policy on the port group
vlanId	vlan ID
vswitchName	vSwitch where portgroup is located
RemovePortGroup:	
__this	host network system reference
pgName	port group name

The VMware documentation of the operations dealing with port groups are listed in Table I. In our model of the operations (Fig. 3), we use the hostname instead of the VMware internal host network system reference to identify a physical machine.

AddPortGroup:

```
AddPortGroup(string hostname, string vswitchName,
  string pgName, int pgVlanId, out node pgNode)
```

This operation adds a new portgroup to a given virtual switch residing on a given host. The production rule tries to match the subgraph consisting of the host and virtual switch, where the names match the parameter values of the rule, e.g. the host's name is matched against parameter 0, denoted as ?0 in Fig. 3a. To this subgraph, a new vswitch node is added that represents the portgroup, to model that the portgroup is an extension of the vswitch. Attributes of the portgroup are set by creating and connecting *data* nodes to the portgroup node based on the input parameters. The node identifier of the newly created portgroup is returned and passed through an *out* parameters as shown below and depicted as !4 in the model (Fig. 3a).

UpdatePortGroup:

```
UpdatePortGroup(string hostname, string pgName,
  string newPGName, int newPGVlanId)
```

Using this operation, an administrator can change the configuration on an existing portgroup. The portgroup is identified by its name, as well as the host where it resides on, and the operation allows to change the portgroup's name and VLAN ID. Changing attributes is modeled as changing the edges to different data nodes based on the input parameters. In order to maintain compatibility with the existing graph model, not only does the portgroup node contain the VLAN ID, but also the associated vport nodes, i.e., virtual switch ports. Therefore, changing the VLAN ID of the portgroup also requires to change the VLAN ID of all virtual ports associated to that portgroup. For this we use the universal quantifier ∀ that applies a sub-rule, given by nodes connected to the quantifier with @ labeled edges, to all its matches. In this case, it updates the vlanId attributes of all matching vport nodes [16] (cf. Fig. 3b).

RemovePortGroup:

```
RemovePortGroup(string hostname, string pgName)
```

In the model of this operation as depicted in Fig. 3c, we delete the vswitch node representing the portgroup given by a name and the host where it resides. Using universal quantification, all vport nodes belonging to the portgroup, i.e., connected through a real edge, are deleted.

B. Dynamic Information Flow Analysis

Bleikertz et al. [4] propose an approach for an information flow analysis on a graph representation of a virtualized infrastructure, in order to detect isolation failures. The approach consists of a set of *traversal rules* that capture how elements in the infrastructure provide isolation, e.g., how VLANs provide network isolation, as well as a graph coloring algorithm to determine information flow. A graph traversal is guided by a set of traversal rules, computes the transitive closure and determines the information flow in the system.

This information flow analysis is limited to a static snapshot of the infrastructure and, thus, unable to deal with its dynamic nature and frequent changes. We show how we can integrate and extend this information flow analysis into our dynamic graph model and, thereby, obtain a dynamic information flow analysis. We express the information flow rules as graph production rules and give an algorithm in *GROOVE*'s control language to derive the information flow analysis from these rules.

The graph-based information flow analysis has three areas, which we describe in the following sections:

1) We compute the *new information flows* of the start graph as initial state of the analysis or when new elements as added.
2) We *adjust existing information flows* for relevant changes in the model, such as when nodes or real edges are removed.
3) We *control the rule application* with control programs to expand the new and adjust rules to a full and terminating information flow analysis, executed after each graph modification.

1) Computing New Information Flows: Computing new information flows is required for an initial flow analysis of a graph or when new elements are added to a graph, e.g., due to the operations transition model. We differentiate between two types of information flow rules: *simple* and *fast-edge* rules. Simple rules are stateless, i.e., they do not perform any modification of graph coloring attribute; stateful rules depend on the color attribute used in graph coloring and are inefficient in graph rewriting. We, therefore, transform stateful rules into fast-edge rules to avoid the stateful rules altogether.

Simple Information Flow Rules: Simple rules can be expressed as graph production rules that match a pair of adjacent nodes with particular types and potentially with conditions on the their attributes. Thus, for each adjacent pair of nodes with a **real** edge between them, the rules create either a *flow* or *noflow* edge. Fig. 4 shows a simple information flow rule that stops information flow between a host and a virtual machine (vmachine) by creating a noflow edge between them, if not already present. This captures the (arguable) trust assumption that no side-channel information leakage exists between virtual machines on the same host.

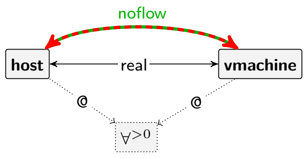


Figure 4: RuleStopVMM: Simple noflow Information Flow Rule.

Information flow edges are conditionally created with a *condition new* rule (cf. §III-A3), i.e., they are only created if they are not already present. This ensures that the information flow analysis will reach a fixed-point when all pairs are connected by either a flow or noflow edge. Furthermore, we use the universal quantifier $\forall^{>0}$ to apply this production rule to all possible matches within one state, in order to reduce the number of states in the state space exploration. It is important that we use the $\forall^{>0}$ operator as it requires at least one match (rather than the $\forall$ operator[1]).

Fast-Edge Information Flow Rules: Formalizing the stateful information flow traversal rules, that is, rules which depend on the attributes of graph elements, are expensive to model in graph transformations as each traversal creates a new graph state. The concept of stateful rules is introduced by Bleikertz et al. [4] to model the information flow between VLAN endpoints, where the information is tagged with a corresponding VLAN identifier at one endpoint, and untagged at the corresponding endpoint. This establishes the information flow such endpoints. We model this behavior using so-called *fast-edges*: They are information flow edges between pairs of nodes that are not necessarily adjacent by a real edge but which are connected through a path.

Fig. 5 shows the production rule for the creation of a fast-edge between two VLAN endpoints in VMware, i.e., the VMware portgroups which are modeled as vswitches with a VLAN identifier and hosted on another vswitch. We conditionally create a flow fast-edge between two distinct portgroups *pg*1 and *pg*2 (pg1 != pg2) if the following conditions hold:

1) VLAN identifiers equality (vlanId == pg2.vlanId).
2) Connectivity of the underlying vswitches, i.e., there exists a flow *path* between them, expressed as flow+.

We again use the universal quantifier $\forall^{>0}$ to apply this rule to all non-empty matches within one state. A similar production rule exists when two portgroups are connected two the same vswitch.

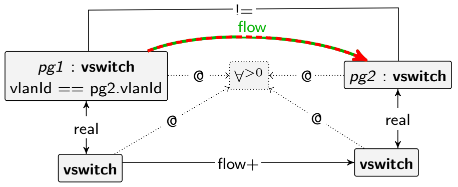


Figure 5: New information flow between two portgroups with the same VLAN ID.

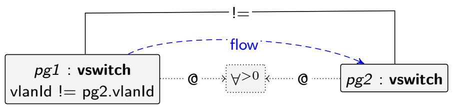
2) Adjust Existing Information Flows: The core of the dynamic information flow analysis lies in the adjustment of existing information flows once the graph changes. We have to handle the following three cases how the graph model may change and we will describe how we adjust the existing information flows due to these changes.

- *Removal of nodes:* The removal of information flow edges that are connected to removed nodes is covered by the underlying graph transformation formalism (*Single Push-Out* [6]) as dangling edges are removed.
- *Removal of real edges:* For each pair of nodes that are no longer connected by a real edge, but still feature an information flow edge, we need to remove the flow edge. This is accomplished by two production rules similar to the simple information flow rules, but with two untyped nodes, an embargo real edge, and the removal of either a flow or noflow edge.
- *Change of nodes' attributes:* The information flow edges that are based on attributes are recomputed if their predicates do not hold anymore. That means, for each information flow rule that introduces an information flow edge based on an attribute condition, we have an adjusting production rule that verifies that the attribute condition still holds; if not, it revokes the information flow edge.

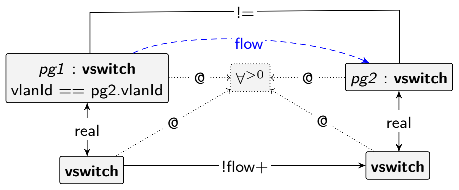
Let us consider the example of the previous VLAN fast-edge production rule, and how the change cases affect the fast-edge and requires adjustment. The VLAN information flow rule depends on two conditions: VLAN equality and vswitches connectivity. The former condition may no longer hold if the VLAN identifier of one

[1]The $\forall$ operator indicates that a rule is applicable even if no matches are found: This would interfere with the termination of our information flow analysis.

portgroup is changed, e.g., due to an UpdatePortGroup operation. The production rule shown in Fig. 6a deletes the flow fast-edge if the VLAN identifier are not equal anymore (vlanId != pg2.vlanId). The second condition may be violated when elements in the graph are deleted and the connectivity of the vswitches is interrupted. Fig. 6b shows a production rule that deletes the fast-edge if the underlying connectivity of the virtual switches is not given anymore, i.e., !flow+, even though the VLAN identifiers are still equal.



(a) Inequality of VLAN IDs.



(b) Connectivity of vswitches is lost.

Figure 6: Adjusting information flows for port groups.

3) Controlling the Rule Application: To complete the information flow analysis for the entire graph, we need to control the application of the rules: *GROOVE* allows for a controlled execution of graph production rules by a control program written in an imperative language. The statements in this language are the rule names, or functions composed of multiple rules, and we have the usual constructs to build sequences, conditions, and loops. We now express the process of information flow analysis as an algorithm in this control language using the previous graph production rules as basic building blocks.

For this task, we differentiate between *explicit* information flow rules, i.e., the *simple* rules that match a particular pair of vertices such as the rule in Fig. 4, and a *default* rule that matches any pair of vertices. The analysis algorithm tries to apply any explicit rules until none is applicable anymore. Then, the default rule is applied until all pairs of vertices have been evaluated. Typically, the explicit rules represent trust assumptions on *isolation* properties of elements in the infrastructure and therefore introduce noflow edges. In contrary, the default rule introduces flow, which means we may perform an over-approximation on the information flows, but also reduce the possibilities of false negatives. The explicit rules we use in this work are designed to be *confluent*, i.e., whenever more than one explicit rule is applicable, it does not matter for the result which one we take first. However, the default rule needs to be applied *after* all explicit rules. After the first information flow analysis, which is based on the simple rules and a default rule, has been completed, we apply the fast-edge rules.

This behavior is captured in the control program of Listing 1. The function info_flow_new introduces information flow edges for an initial graph or for new elements in a graph. The choice operator lets *GROOVE* pick any of the applicable explicit rules, such as RuleStopVMM shown in Fig. 4. As we said before, for the final result is does not matter which one is picked if more than one is applicable. The try ... else Default ensures that whenever none of the explicit rules is applicable anymore, then the Default rule is applied. The alap finally ensures that this process is repeated *as long as possible*, i.e., we finish if neither explicit rules nor the default rule is applicable anymore. This whole process must eventually terminate, because each application of a rule for a pair of connected nodes reduces the number of unevaluated pairs of nodes. This is because information flow edges are added as *conditionally new* ensuring that a rule can only be applied once to every connected pair. Afterward, the fast-edge rules are applied, where RuleFlowNewVlanFastEdge1 is the rule shown in Fig. 5 and RuleFlowNewVlanFastEdge2 is the similar rule when two portgroups are connected to the same vswitch.

```
function info_flow_new() {
  alap try {choice RuleStopVMM;
          ...
          or RuleN;
        }
     else RuleFlowDefault;

  try RuleFlowNewVlanFastEdge1;
  try RuleFlowNewVlanFastEdge2;
}
```

Listing 1: Control Function for New Information Flows.

In order to cope with the dynamic behavior, the information flow analysis needs to perform the information flow adjustments based on the previously introduced production rules. For adjusting existing information flow upon changes in the graph, the function info_flow_adjust of Listing 2 applies the production rules to remove dangling information flow edges, followed by the production rules that verify that the conditions for the VLAN fast-edges still hold, and if not remove the fast-edges.

```
function info_flow_adjust() {
  RemoveDanglingFlowEdge;
  RemoveDanglingNoFlowEdge;
  try RuleFlowAdjustVlanFastEdge1;
  try RuleFlowAdjustVlanFastEdge2;
}
```

Listing 2: Control Function for Adjusting Flows.

C. Security Policies

The final part of our unified model deals with the formalization of security policies. We follow the approach of the policy language *VALID* [2], which allows to specify security properties on the topology and configuration of an infrastructure cloud. We express the security policies as *attack* states, i.e., a state of the topology or configuration of the infrastructure cloud that violates the desired security property. Instead of verifying that a security property holds for the entire infrastructure, we try to find a violation.

Whereas *VALID* policies try to match a set of facts, we can also express such policies as graph production rules in *GROOVE*. Such production rules only try to match a particular subgraph, which constitutes an attack state, and may have additional constraints on the subgraph.

As an example, we consider the security policy of *zone isolation* [2], or its corresponding attack state of zone isolation breach. This policy introduces the concept of security *zones* that the virtual machines belong to, and defines an *isolation breach* to be any information flow between virtual machines that belong to different zones. The concept of zones is part of the definition of the security policy. We can model them by introducing new nodes for each zone with an attribute zone.name and edges from zones to virtual machines machines to model that a zone contains a virtual machine. Typically, every virtual machine belongs to exactly one zone, except for machines that act as gateways between two zones, such as virtual firewalls.

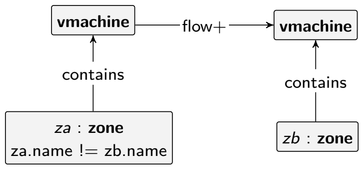


Figure 7: Zone Isolation Breach Rule in *GROOVE*.

A breach of zone isolation is defined by the graph production rule in Fig. 7. Here, we try to match a pair of vmachines that are connected by an information flow path and that belong to different zones, i.e., given by different zone name attributes. Verifying the compliance of an infrastructure against a security policy by trying to match a graph production rule that expresses a policy violation will be described in §IV-A3.

D. Lessons Learned

Modeling a complex system such as a virtualized infrastructure and its management operation using graph transformations provided us the following insights.

Operations that change the configuration and topology of virtualized infrastructures are intuitively modeled as graph transformations. Universal quantifications are a useful method to update many infrastructure elements within one operation atomically. Otherwise, the same behavior has to be modeled using multiple separated production rules, which is inefficient due to an increased number of states, and it increases the complexity of the model. Parametrized production rules are essential to capture the also parametrized cloud management operations.

In order to keep the model simple and therefore suitable for automated analysis, it is essential to focus on a subset of operations, i.e., which modify the topology and configuration. Furthermore, many operations use parameters that do not impact our security analysis and model, therefore can be omitted in the operations model.

Complex operations such as the creation of a VM modify many different aspects of the infrastructure, e.g., besides creating a VM it also creates and attaches virtual storage and network devices. Ideally, such operations are divided into sub-operations and are modeled in a modular way, in order to reduce the overall complexity. However, current limitations in *GROOVE* when combining modularization with parametrization inhibits this modular modeling. This is not a fundamental problem of the graph transformation system, rather a limitation in its implementation.

A key insight for the modeling techniques employed for the production rules was to keep the number of states in the graph transformation system as low as possible through the means of using universal quantifiers. Instead of applying a production rule every time for a match, which would result in a new state for each match, we apply the rule for all matches at the same time using the universal quantifier, which only results in one new state. This is particularly important for the information flow analysis, where rules will have many matches, and therefore could produce many new states.

IV. Automated Analysis and Applications

Our system aims at an automated analysis of configuration and topology changes in virtualized infrastructures. Its architecture obtains all the necessary inputs for the analysis and invokes *GROOVE* as the graph transformation and analysis tier. We introduce the architecture and integration first and subsequently establish two application scenarios for change management (§IV-B) and auditing of configuration changes at run-time (§IV-C). As future work, we propose the run-time enforcement of security policies and the mitigation of misconfigurations. This is achievable with our system, and we outline challenges for that (§IV-D).

A. Architecture and Integration

We depict the system architecture in Fig. 8. The main components are i) the *Configuration Discovery &*

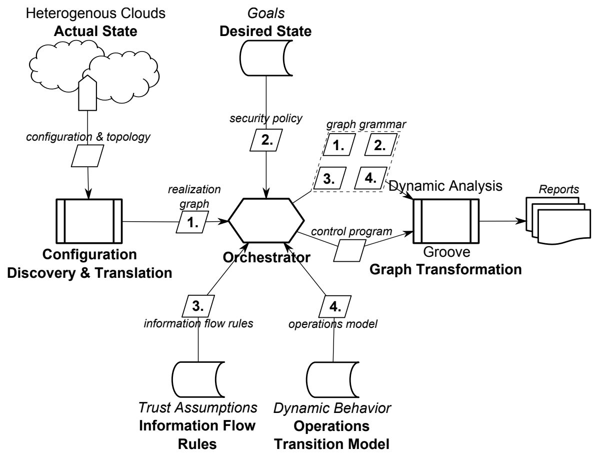


Figure 8: System Architecture

Translation on the left, which extracts the configuration of the virtualized infrastructure and constructs the graph model; ii) the *Orchestrator* in the middle, which integrates all required inputs and prepares the graph grammar for analysis; iii) and the *Graph Transformation* on the right, which employs *GROOVE* for the analysis given the prepared graph grammar of the orchestrator.

1) Preparing the Graph Grammar: The *Orchestrator* obtains the inputs for the analysis and prepares the graph grammar for *GROOVE*. The graph grammar needs to be encoded in *GXL* [20], a XML-based graph format. The following inputs are required:

Start Graph: The orchestrator creates a graph model of the virtualized infrastructure, using the approach outlined in §II-B. We employ a graph simplification algorithm in order to reduce the size of the graph and improve the performance of the analysis (cf. §IV-A2). *GROOVE* expects a certain graph structure, i.e., directed edges and nodes' attributes encoded as self edges with the attributes as edge labels. Our case study on zone isolation requires to annotate which virtual machines belong to which security zones. We realize this annotation by introducing zone nodes and establishing edges between zones and virtual machines (cf. §III-C) based on a security policy specification given by the user.

Security Policy: Policies, and their attack states, are modeled within *GROOVE* as graph production rules. A user either selects an existing policy, e.g., the zone isolation policy, or creates new policies using the graphical modeling environment of *GROOVE*.

Information Flow Rules: The rules represent the trust assumptions on isolation capability of components of the virtualized infrastructure. They are also modeled as production rules. Typically, a set of best-practice information flow rules is used, but a user can also modify or extend the rules using the graphical modeling environment.

Operations Transition Model: We model management operations specific to a virtualization platform in *GROOVE*. In general, we do not expect that users need to modify the operations transition model, once all relevant operations have been incorporated. Finally, the orchestrator obtains from the user the desired changes to be analyzed and outputs a *Control Program* for *GROOVE*. A sample control program is shown in Listing 3 that contains an UpdatePortGroup operation besides the information flow analysis.

```
info_flow();

UpdatePortGroup("host1.domain.tld", "portgroup-23", ↩
    123, "portgroup-23");

info_flow();
info_flow_adjust();
```

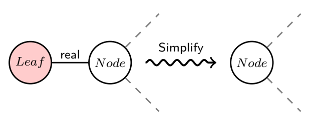
Listing 3: Produced Control Program Produced.

2) Simplification of the Start Graph: Similar to star-mesh transformations employed for the simplification of electrical networks, we perform a simplification algorithm to reduce the size of the *GROOVE* start graph. Our graph simplification algorithm preserves nodes, which are used in the analysis, and their inter-connectivity.

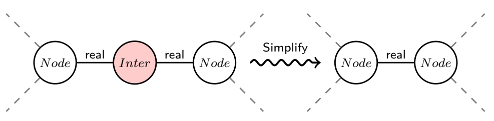
We extract a list of node types that need to be preserved from the information flow rules, operations model, and security policies in the graph grammar. The simplification algorithm needs to preserve nodes of these types, in order to not change the analysis results. We define a *candidate* as a node in the graph that has a type not occurring in the preserved types list. A candidate can be removed from the graph while preserving the connectivity of the graph in the following three ways.

- *Removal of a leaf node* (Fig. 9a): Candidates that are leaf nodes can be simply removed as they do not contribute to the connectivity between preserved type nodes.
- *Removal of an intermediate node* (Fig. 9b): Candidates that are intermediate nodes can be removed, but their neighbors have to be connected by an edge, in order to preserve connectivity.
- *Star-Triangle (Y-Δ) transformation* (Fig. 9c): The transformation from a star to a triangle topology does not reduce the number of edges, but decrements the number of nodes by removing the star node.

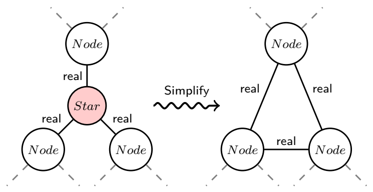
For candidates with a degree higher than 3 the star-mesh transformation does not reduce the number of edges or nodes anymore. We apply the simplification rules until no candidates with a degree lower or equal than 3 are available. The simplification does not tamper the analysis results because it removes only nodes that are not relevant for the analysis, i.e., they do not occur in information flow rules, the operation model, nor in the security policies. Furthermore, connectivity is preserved while removing nodes during the graph simplification.



(a) Leaf Node



(b) Intermediate Node



(c) Star Node

Figure 9: Graph Simplifications for Nodes of Different Degrees.

3) Graph Transformation Tier: We employ *GROOVE* for the graph transformation and analysis once the orchestrator prepared the graph grammar and control program. Of particular interest is the violation of a given security policy, or, if an attack state will be reached for the desired changes to the infrastructure. *GROOVE* provides the concept of *acceptors* that indicate when the graph exploration has found a *result state*. One can limit the number of result states that the graph exploration should find, in order to trigger a termination once an acceptor fires. In our case, the acceptor will be set to the *invariant* acceptor with the security policy rule, which fires when the given rule is applicable, and we limit the number of result states to one. In this case, the graph exploration can be stopped once an attack state for the policy of zone isolation is reached, which is indicated with the following acceptor parameter `inv:PolicyIsolationBreach` (cf. Fig. 7).

Furthermore, each result state is an instance of the infrastructure where the policy is violated. The match of the policy rule in each result state provides an instance of the policy violation, e.g., a pair of virtual machines that violate the isolation policy. As the *GROOVE* command line do not export such matches, we realized a wrapper around *GROOVE* to obtain them.

GROOVE supports multiple different graph exploration strategies. In our case, we employ a *linear* exploration strategy, because i) our simple and explicit information flow rules (which are used with the choice operator) are confluent and ii) our rule application is strictly guided by the control program. Analyzing the interleaving of all the desired changes through a full state space exploration could be interesting, but at the cost of a large state space and therefore negatively impacted performance of the analysis.

B. Change Plan Analysis

Change plans can help to improve the quality of IT infrastructures as changes are properly documented and can be evaluated by another party, e.g., another administrator or in our case by an automated tool. Change management is often employed as part of IT infrastructure operation workflows and processes. In our case, an administrator drafts a sequence of desired changes in the form of cloud management operations, which will eventually be provisioned to the virtualized infrastructure. The crucial question is: Will a proposed change render the infrastructure insecure?

To answer this question, the administrator submits the change plan to our system, which constructs a control program for *GROOVE* containing the desired changes. *GROOVE* performs a graph exploration, i.e., it applies the changes to the graph model of the infrastructure and verifies the resulting infrastructure state against the desired security policies. By that, the tool can establish a what-if analysis and determine what security impact the intended changes will have on the infrastructure.

If the new graph model obtained from the application of the changes violates the security goals, the tool notifies the administrator to reject the proposed change plan and provides the analysis output of the matched policy violation as diagnosis. Otherwise, the tool returns that the intended changes are compliant with the security goals, after which the administrator can provision the changes to the infrastructure.

The example of a change plan in Listing 4 specifies the addition and updating of virtual network elements in the infrastructure model and allows us to check whether this additional or modified network connectivity will violate a security policy, e.g., the zone isolation. It could be that the final UpdatePortGroup causes a violation. Regarding the languages syntax, return values in *GROOVE*'s control language are denoted as out parameters.

```
AddVirtualSwitch("host1", "vswitch2");
node PG;
AddPortGroup("host1", "vswitch2", "portgroup4", 23, ↩
    out PG);
string Dev;
AddVirtualNic("host1", "portgroup4", "127.0.0.1", ↩
    "00:FF:00:FF:00:FF", out Dev);

UpdateVirtualNic("host1", Dev, "127.0.0.2", ↩
    "00:FF:00:FF:00:AA");
UpdatePortGroup("host1", "portgroup4", 24, ↩
    "portgroup4");
```

Listing 4: Change Plan for Adding and Updating Virtual Network Components.

C. Run-time Audit of Configuration Changes

Run-time audit of configuration changes expands on the principles of the change plan analysis in §IV-B. Whereas change planning requires the administrator to devise the changes in advance and have them checked by our system statically, the run-time audit intercepts change requests dynamically at a proxy and checks them as they occur. The idea of the run-time auditing is to establish accountability for administrator actions: The administrator's configuration changes are validated against the security policy and the results of these checks entered into the audit logs along with the administrator's username and the committed commands.

We introduce an *authorization proxy* as wrapper of the cloud administration API, which acts as auditor of configuration changes, and employ our system as part of the policy decision mechanism.

The authorization proxy is a reverse HTTPS proxy in front of the otherwise shielded infrastructure manager. It intercepts management operations and inspects them for the auditing. The communication in front of the manager is usually standardized: VMware and Amazon EC2 management operations are SOAP-based, whereas OpenStack is REST-based. These formats are easily inspected. In addition, the proxy tracks session states derived from the the user-login and the infrastructure manager session cookie, to distinguish sessions of multiple administrators interacting with the infrastructure manager concurrently.

The Policy Decision Point (PDP) of the authorization proxy translates intercepted management operations into a change plan in the *GROOVE* control language. We have translation modules for all covered operations. For instance, from an UpdatePortGroup operation the proxy extracts the host, identifying portgroup name, new VLAN identifier, as well as new portgroup name, and generates a control program line such as the following one similar to Listing 4:

```
UpdatePortGroup("host1", "portgroup4", 24, ↩
    "portgroup4");
```

The Policy Decision Point delegates the change plan analysis to our system. The system notifies the administrator of the result of the security analysis and produces an audit trail for accountability. Furthermore, a feedback mechanism directs administrators' behavior towards considering configuration changes carefully.

D. Run-time Mitigation and Enforcement

As future work, we pursue the following research hypothesis: Run-time analysis can be used for automated mitigation of misconfigurations and enforcement of a security policy. With proper enforcement mechanisms in place, it allows to protect virtualized infrastructures from malicious adversaries. We establish the invariant that all configurations changes are only accepted if the security analysis returns that no security policy is violated.

Observe that there need to be additional security mechanisms in place to defeat malicious adversaries, even if the analysis functionality is unaffected by that. To protect against those, the infrastructure needs to be modified to enforce sole access through the management interface, and need to prevent circumvention approaches, such as a direct SSH log-in to the physical hosts. Also, the security validation needs to be mandatory for all management operations.

Again, the authorization proxy intercepts operations and delegates the security analysis to our system. This time it acts as Policy Enforcement Point (PEP): The intercepted operations are accepted, if the operations fulfill the policy; otherwise, they are rejected. The authorization proxy refrains from forwarding the management operation in the reject case, i.e., they are not deployed in the actual infrastructure. It signals an error back to the administrator client, including the policy violation as data for diagnosis. We have pursued this direction and extended the authorization proxy with this capability, but it requires future work on virtualized infrastructures and provisioning systems to be viable:

First, operations of multiple administrators interleave concurrently. These operations are not atomic: Whereas operation such as UpdatePortGroup seem instant to administrators, they are not guaranteed to be atomic. Tasks are asynchronous by design and take time to complete. Thus, operations can interfere with each other.

Second, the runtime mitigation might block management operations. Soft blocking occurs because of the delay the analysis adds and may be precarious if the expected time between management operations is smaller than the expected analysis time. Hard blocking occurs if the authoriztion proxy rejects a management operation to enforce the security policy: The administrator might need to override the security policy in an emergency to prevent a catastrophic failure.

Still, run-time mitigation is an important topic of future work poised to increase the security assurance of virtualized infrastructures in face of malicious adversaries.

V. Evaluation and Discussion

We are evaluating our system with regard to performance, e.g., how does it cope with large virtualized infrastructures, and its effectiveness in analyzing changes for misconfigurations, i.e., can it detect faulty configuration changes and differentiate them from safe ones. We consider two VMware-based virtualized infrastructures for this matter: one is a small laboratory infrastructure, and the other is a mid-sized production one with around 1400 VMs.

For each infrastructure, we evaluate two change plans, where one specifies a configuration change that violates a security policy, and the other specifies a safe change. The combination of infrastructures and change plans establishes four scenarios, for which we measure the overall analysis time as well as the time for each analysis step, and the effectiveness of detecting the change plans with the faulty changes. We focus the evaluation on the change plan analysis, as the run-time audit boils down to generating and analyzing a change plan.

A. Lab and Production Cloud Scenarios

We evaluate our system for four scenarios, based on two different virtualized infrastructures – a laboratory and production one – and for each a safe and faulty change plan. The lab infrastructure consists of 4 hosts, 16 VMs, of which 2 are in a *production* security zone, 3 in a *test* zone, and 11 are currently unassigned. The graph model consists of 210 nodes and 548 edges, and in its simplified form 101 nodes and 310 edges. The production infrastructure contains 60 hosts, around 1400 VMs, and overall five security zones. The corresponding graph model contains 23579 nodes and 61564 edges, and the simplified graph 9576 nodes and 32902 edges. In our time measurements we use the simplified graph models if not otherwise stated. We consider the security policy of *zone isolation* (cf. §III-C), although in the production cloud case not all zones are strictly isolated from each other, and we modify the policy to ensure isolation for two specific zones.

The desired change that is specified in the change plans is an UpdatePortGroup operation that changes the VLAN configuration. In a *faulty* change plan, the new VLAN ID of a portgroup that hosts VMs of a zone A is already used by a portgroup that hosts VMs of a zone B. This leads to an undesired information flow between the two zones and entails a potential zone isolation breach, if the two zones need to be isolated from each other. A *safe* change plan would change the VLAN ID to an unused one or one that is already used by portgroups hosting VMs of the same zone.

B. Methodology

In order to determine the effectiveness of our system, we compare its analysis results with our knowledge of the change plans, i.e., our system needs to report a policy violation for a faulty change plan and should not report a violation for a safe one.

Regarding the performance of our system, we measure the overall time required to complete the change plan analysis for our scenarios. Furthermore, we measure the time for the individual analysis steps, such as performing the information flow analysis, applying the changes, and the policy matching. We run our measurements in a VM (on VMware ESXi 5.1) with 12GB of memory, 12 CPU cores @ 2.4GHz, Linux 3.8 64bit with IBM JVM 1.7, and *GROOVE* version 4.8.7. However, *GROOVE* mostly utilizes 1-2 cores and memory usage is below 4GB. For each measurement, *GROOVE* is run 2 times for warming up any caches and followed by 10 runs for the actual time measurement, for which we calculate the mean and standard deviation.

In the case of measuring the *complete* analysis time, we execute *GROOVE* with the policy breach rule as the *invariant* acceptor, i.e., the graph exploration will terminate once the policy breach rule is applicable. Thereby one instance of a policy violation is identified, although it can be configured to identify all instances.

Measuring the individual analysis steps is more challenging, as *GROOVE* does not provide any methods for profiling or timing their control programs, and we refrained from modifying the code. Therefore, we pursue an *incremental* approach, where we gradually enable more steps in the analysis, measure the time and calculate the difference to the previous measurement, where the current step was not enabled. This provides us with a relative time measurement for each analysis step and we measure the following steps:

- *Start*: Loading of JVM, loading of grammar.
- *Init Info Flow*: Initial information flow analysis.
- *Change Op*: The operation that induces a change.
- *Adjust Info Flow*: Updating information flows.
- *Policy Match*: Matching the policy rule.

In this case, we execute *GROOVE* with the *final* acceptor, i.e., *GROOVE* terminates when no more rules are applicable. By this we ensure that all steps are fully executed and prohibit an early termination as it is in the case for the policy invariant acceptor.

C. Results and Discussion

In our experiments for the different scenarios, our system successfully found a policy violation for the *faulty* change plans, and reported the policy compliance for the *safe* scenarios. Regarding its performance, Table II contains the time measurements for the four scenarios in absolute, as well as in relative form based on our incremental measurements. Observe that negative relative times for the incremental approach are slight inaccuracies due to the imprecision of the measurement method for short analysis steps.

Comparing the time measurements for the *Complete* analysis between a *safe* and *faulty* change plan shows that the time for a faulty change plan analysis can be significantly shorter, as it is shown for the production infrastructure. This effect is caused by the policy invariant acceptor of *GROOVE* with a limitation to one result state, which allows an early termination of the analysis once a policy violation is found. In the faulty change plan

Scenario	**Complete**	**Incremental** (with cumulative absolutes)				
		Start	*Init Info Flow*	*Change Op*	*Adjust Info Flow*	*Policy Match*
Lab, Safe	5.61 ± 0.25	5.41 ± 0.22	5.6 ± 0.2	5.57 ± 0.22	5.6 ± 0.2	5.6 ± 0.16
relative			0.19 ± 0.29	-0.04 ± 0.29	0.03 ± 0.29	0.0 ± 0.26
Lab, Fault	5.74 ± 0.14	5.26 ± 0.29	5.66 ± 0.08	5.6 ± 0.09	5.7 ± 0.09	5.64 ± 0.29
relative			0.39 ± 0.3	-0.06 ± 0.12	0.1 ± 0.13	-0.05 ± 0.3
Production, Safe	154.71 ± 6.91	19.6 ± 1.13	48.58 ± 1.19	50.26 ± 1.26	154.32 ± 7.88	152.15 ± 5.54
relative			28.98 ± 1.64	1.68 ± 1.73	104.06 ± 7.98	-2.17 ± 9.63
Production, Fault	69.53 ± 1.71	19.7 ± 0.67	50.04 ± 1.92	48.85 ± 1.54	153.35 ± 4.73	164.78 ± 5.49
relative			30.33 ± 2.04	-1.18 ± 2.47	104.5 ± 4.98	11.43 ± 7.25

Table II: Absolute and Relative Time Measurements (in seconds) for Change Plan Analysis with *GROOVE*.

Scenario	**Simplified**	**Non-Simplified**
Production, Safe	154.71 ± 6.91	1026.24 ± 179.86
Production, Fault	69.53 ± 1.71	181.1 ± 6.45

Table III: Time Measurements (in seconds) for Change Plan Analysis with Simplified and Non-Simplified Host Graphs.

case, *GROOVE* terminates once one policy violation is found, whereas for the safe change plan, all possibilities for a policy violation needs to be checked but none is found, which of course requires more time. In practice, the policy invariant acceptor will be used for the change plan analysis, in order to allow an early termination once a policy violation is found. However, one could also run the analysis to find *all* policy violations, instead of terminating once one is found.

The incremental measurement for the analysis step of adjusting the information flows is surprisingly high compared to the initial flow analysis. Investigation of this performance issue pointed to the adjust information flow for vswitches connectivity of Fig. 6b. We suspect that the root cause is an inefficient implementation in *GROOVE* for evaluating the negative path expression !flow+ compared to the positive one of Fig. 5.

The overhead of starting the JVM and loading the grammar, up to the point where the control program is executed, can be relatively large, especially for small infrastructures. For the *Lab* scenarios, the other analysis steps are negligible compared to the start overhead. For the *Production* scenarios, the overhead is mainly induced by loading the start graphs from the grammar, which entails a XML deserialization. Change operation in all cases negligible. Operations are well-defined, i.e., they change a specific part of the graph with a single match and transformation. This makes them simple and cheap to apply. In contrary, adjusting the information flow is the most expensive analysis step as we have to verify that the conditions for the VLAN fast edges still hold.

The impact of the graph simplification on the analysis of a change plan for the production infrastructures is significant, in particular for a safe change plan. Table III shows the time measurements for the production infrastructure scenarios based on a *simplified* and *non-simplified* host graph. In the faulty change plan scenario, the analysis based on a non-simplified graph is slower by a factor of 2.6 ± 0.1. In the case of a safe change plan, the required analysis time increases by a factor of 6.6 ± 1.2 between the analysis of a simplified and non-simplified graph. The time required for performing the simplification on the production host graph is 76 seconds. The costs for the simplification are already amortized for the first analysis of a change plan in the production scenarios, and observe that it has to be done only once per graph. We observed a high memory consumption of almost all available host memory of 12GB for the non-simplified graph and safe change plan scenario. For practical applications, an analysis time in the area of 17 minutes for a safe change plan, which is probably the majority case, would not be acceptable. Therefore, graph simplification is a valuable tool to provide reasonable performance for change plan analysis.

D. Lessons Learned and Further Optimizations

The performance of the analysis is largely impacted by two factors: the size of the start graph and the modeling techniques used for the production rules. Our graph simplification algorithm (see §IV-A2) reduces the size of the start graph, which has a significant impact (cf. Table III) on the analysis time. We already discussed lessons learned from the modeling in §III-D and a key insight is to reduce the number of produced states by employing the universal quantifier operator in the rules.

Although the performance of our system is reasonable for the two current application scenarios of change plan analysis and run-time audit, for the future run-time enforcement scenario further optimizations would be required. Optimizations in the modeling, such as creating a star topology for the VLAN fast edges, would probably improve the performance, but at the cost of a more complicated modeling. Therefore, we favor in this work a trade-off between clarity in the modeling and reasonable performance. Furthermore, as our results show, the start time for the analysis can be relatively large, especially for small infrastructures. One could address this problem by running *GROOVE* in a "daemon"-mode, where it is running with the graph grammar already loaded and evaluating different change plans.

E. Security Discussion

The effectiveness of our analysis is impacted by the different parts of our model. As state in §II-C, we require an authentic view of the topology and configuration of the infrastructure as well as a faithful model of it. The information flow rules represent the trust assumptions on isolation properties and a wrongly selected set of rules may alter the effectiveness of the analysis. We adopted a set of rules which have been discussed and argued for in [4]. The specification and selection of the security policy depends on the individual application environment. Our system only finds attack states for configuration changes, and does not provide a security proof. Regarding the operations transition model, it is not possible to prove the correctness of the model, but we employed a systematic approach to obtain such a model.

Obtaining detection rates for our system is not possible at the moment, as reference data sets are not available. This is a similar situation as for the first intrusion detection systems, which also lacked reference data sets at the beginning. We will approach this challenge by deploying our system for practical deployments as future work.

One potential attack vector is the complexity of SOAP that can be exploited to break the proper inspection of the web service requests within the authorization proxy. Somorovsky et al. [18] show successful attacks against the authentication of SOAP-based cloud management interfaces. However, this potential attack vector is rooted in a software vulnerability of parsing SOAP messages, not a potential inherent security flaw in our architecture.

VI. Related Work

We discuss our work with related research in a top-down approach focusing first on the application scenarios and overall goals of our system, followed by comparisons on the modeling approaches and underlying concepts.

Formal Approaches to System Management: A model-based approach for configuration management has been proposed in [14] which formalizes network configurations in first-order logic (FOL) and employs *Alloy* [13] for model finding. Both theirs and our approach tackle configuration error detection and mitigation of misconfigurations, whereas theirs even support configuration synthesis and error fixing. A difference between the approaches is that theirs is currently limited to network configurations, whereas our underlying model supports the entire virtualized infrastructure, i.e., compute, network, and storage. A further difference lies in the specification of requirements and configuration changes. In their approach, they are formalized in FOL, whereas in our approach policies and changes are modeled as graph transformations, which are applied from an imperative control program. We believe that change plans specified in an imperative language with statements resembling the cloud management operations may be more intuitive to administrators than FOL due to their background in scripting languages, rather than logic. Furthermore, our application scenario of run-time audit automatically creates change plans based on the interactions.

Policy-based System Administration: Policy-based system administration tools, such as the popular CFEngine [5], allow the specification of a *desired state* for server configurations. Further, such tools enable the monitoring and implementation of the configuration to reach and maintain such a desired state. However, these tools mostly focus on operating system' and applications' configurations, and less on the network and storage infrastructure. Our system provides a complementary approach as we do not focus on the configuration of virtual machines, i.e., servers, but the configuration and topology of the underlying virtualized infrastructure, which spans computing, networking, and storage. *PoDIM* [8] is a high level language for configuration management that separates an administrator's intentions from low-level changes. However, it mainly focuses on the configuration of individual systems, although with certain cross-machines constraints, and importantly it lacks the analysis of the changes. One could combine PoDIM with our system, where an administrator specifies high-level configuration changes in PoDIM, and the translated low-level changes of the virtualized infrastructure are analyzed by our system.

Security Policy Verification for Virtualized Infrastructures: Existing work in this field covers the areas of policy specifications [2] and the analysis of virtualized infrastructures [3], [4] as well as discuss its implications in detail. However, these efforts lack the ability to handle dynamic behavior. With our system we close this gap by introducing the first formal model of changes induced by management operations in virtualized infrastructures.

Graph Transformations for Security Applications: Graph transformations and in particular *GROOVE* have found applications in other security-related scenarios. A security case study has been presented in [12] that deals with the graph-based modeling of physical and digital environments based on the *Portunes* [9] framework. Their overall approach is similar to ours, where the environment is given as a graph that contains elements of spatial, physical, and digital kinds. Graph transformations are used to formalize actions of entities in the environment graph, e.g., to express mobility of entities. A security policy is specified as an attack state and the graph exploration of *GROOVE* tries to find an environment graph where the attack state is satisfied by performing available actions for entities in the environment.

VII. Conclusion and Future Work

In this work, we tackle the problem of misconfigurations in virtualized infrastructures. Our solution consists of a practical security system that employs a formal model of cloud management operations in order to proactively assess their security impact. Building our modeling efforts upon graph transformation, we offer a unified approach in an intuitive and expressive form. We propose two application scenarios for our system – change planning and run-time auditing – and evaluate our system for virtualized infrastructures in laboratory and production environments, which yields performance appropriate for practical scenarios.

As future work, we will pursue the run-time enforcement direction already outlined in § IV-D. Besides the challenges already mentioned there, we intend to further improve the performance of the analysis by *GROOVE*. Potential optimizations are described in §V-D, yet come at the cost of a more complicated modeling. Furthermore, a large part of future work consists of running user studies and deploying our system in production environments. This paper offers a first step for and gives direction to tackling the problem of misconfigurations in virtualized infrastructures; we demonstrated the feasibility and performance even for production environments. Evaluating the system with practitioners will provide further insights with regard to usability and effectiveness in practical real-life environments.

Acknowledgments

This research has been partially supported by the TClouds project[2] and TREsPASS project[3] funded by the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement number ICT-257243 and ICT-318003.

References

[1] Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on 1*, 1 (jan.-march 2004), 11 – 33.

[2] Bleikertz, S., and Gross, T. A Virtualization Assurance Language for Isolation and Deployment. In *IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY'11)* (Jun 2011), IEEE.

[3] Bleikertz, S., Gross, T., and Mödersheim, S. Automated Verification of Virtualized Infrastructures. In *ACM Cloud Computing Security Workshop (CCSW'11)* (Oct 2011), ACM.

[4] Bleikertz, S., Gross, T., Schunter, M., and Eriksson, K. Automated Information Flow Analysis of Virtualized Infrastructures. In *16th European Symposium on Research in Computer Security (ESORICS'11)* (Sep 2011), Springer.

[2]http://www.tclouds-project.eu

[3]http://www.trespass-project.eu

[5] Burgess, M. A Site Configuration Engine. *Computing Systems 8*, 2 (1995), 309–337.

[6] Corradini, A., Ehrig, H., Heckel, R., Korff, M., Löwe, M., Ribeiro, L., and Wagner, A. *Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations.* Vol. 1 of Rozenberg [17], 1997, ch. Algebraic Approaches to Graph Transformation - Part II: Single Pushout Approach and Comparison with Double Pushout Approach, pp. 247–312.

[7] CSA. Top threats to cloud computing v1.0. Tech. rep., Cloud Security Alliance (CSA), mar 2010.

[8] Delaet, T., and Joosen, W. PoDIM: A Language for Gigh-Level Configuration Management. In *Proceedings of the 21st conference on Large Installation System Administration Conference* (Berkeley, CA, USA, 2007), LISA'07, USENIX Association, pp. 21:1–21:13.

[9] Dimkov, T., Pieters, W., and Hartel, P. Portunes: Representing Attack Scenarios Spanning through the Physical, Digital and Social Domain. In *Proceedings of the 2010 joint conference on Automated reasoning for security protocol analysis and issues in the theory of security* (Berlin, Heidelberg, 2010), ARSPA-WITS'10, Springer-Verlag, pp. 112–129.

[10] ENISA. Cloud computing: Benefits, risks and recommendations for information security. Tech. rep., European Network and Information Security Agency (ENISA), nov 2009.

[11] Garfinkel, T., and Rosenblum, M. When Virtual is Harder than Real: Security Challenges in Virtual Machine Based Computing Environments. In *HOTOS'05: Proceedings of the 10th conference on Hot Topics in Operating Systems* (Berkeley, CA, USA, 2005), USENIX Association, pp. 20–20.

[12] Ghamarian, A. H., de, M. M., Rensink, A., Zambon, E., and Zimakova, M. Modelling and analysis using GROOVE. *International Journal on Software Tools for Technology Transfer (STTT)* (March 2011).

[13] Jackson, D. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol. 11* (April 2002), 256–290.

[14] Narain, S. Network Configuration Management via Model Finding. In *Proceedings of the 19th conference on Large Installation System Administration Conference - Volume 19* (Berkeley, CA, USA, 2005), LISA '05, USENIX Association, pp. 15–15.

[15] Oppenheimer, D., Ganapathi, A., and Patterson, D. A. Why do internet services fail, and what can be done about it? In *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems - Volume 4* (Berkeley, CA, USA, 2003), USITS'03, USENIX Association.

[16] Rensink, A., and Kuperus, J.-H. Repotting the geraniums: on nested graph transformation rules. In *Graph transformation and visual modelling techniques, York, U.K.* (2009), A. Boronat and R. Heckel, Eds., vol. 18 of *Electronic Communications of the EASST*, EASST.

[17] Rozenberg, G., Ed. *Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations*, vol. 1. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997.

[18] Somorovsky, J., Heiderich, M., Jensen, M., Schwenk, J., Gruschka, N., and Lo Iacono, L. All Your Clouds are Belong to us – Security Analysis of Cloud Management Interfaces. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop* (New York, NY, USA, 2011), CCSW '11, ACM.

[19] VMware. vSphere 5.0 API Reference, Aug 2011. http://pubs.vmware.com/vsphere-50/topic/com.vmware.wssdk.apiref.doc_50/right-pane.html.

[20] Winter, A., Kullbach, B., and Riediger, V. An Overview of the GXL Graph Exchange Language. In *Revised Lectures on Software Visualization, International Seminar* (London, UK, UK, 2002), Springer-Verlag, pp. 324–336.

Notes