# Cloud Radar: Near Real-Time Detection of Security Failures in Dynamic Virtualized Infrastructures

Sören Bleikertz Carsten Vogel
IBM Research - Zurich
{sbl,ten}@zurich.ibm.com

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

## ABSTRACT

Cloud infrastructures are designed to share physical resources among many different tenants while ensuring overall security and tenant isolation. The complexity of dynamically changing and growing cloud environments, as well as insider attacks, can lead to misconfigurations that ultimately result in security failures. The detection of these misconfigurations and subsequent failures is a crucial challenge for cloud providers—an insurmountable challenge without tools.

We establish an automated security analysis of dynamic virtualized infrastructures that detects misconfigurations and security failures in near real-time. The key is a systematic, differential approach that detects changes in the infrastructure and uses those changes to update its analysis, rather than performing one from scratch. Our system, called *Cloud Radar*, monitors virtualized infrastructures for changes, updates a graph model representation of the infrastructure, and also maintains a dynamic information flow graph to determine isolation properties. Whereas existing research in this area performs analyses on static snapshots of such infrastructures, our change-based approach yields significant performance improvements as demonstrated with our prototype for VMware environments.

## 1. INTRODUCTION

Infrastructure clouds are rapidly and dynamically changing systems due to self-service provisioning and on-demand scalability. Tenant as well as provider administrators frequently adapt the configuration of the sub-system they control, constituting in dynamic changes for the entire configuration. These changes may create security vulnerabilities with respect to the tenants' individual or the provider's overall security policies. According to studies by ENISA [5] as well as CSA [4], isolation failures as well as operational complexity leading to misconfiguration and security failures are among the top threats in cloud computing. Those vulnerabilities can be introduced as a non-deliberate fault or be the deliberate act of an insider attacker, affecting all the pillars of infrastructure clouds: computing, networking, and storage. While misconfiguration of network components (e.g., subnets and VLAN IDs) are recognised as the faults of isolation breaches, unwanted co-location of computing resources [11] or misconfigured storage isolation have been observed, too. While configurations of multi-tenant infrastructure clouds are complex in themselves, overseeing the security consequences of many configuration changes by multiple administrators can easily be beyond the grasp of human operators.

Indeed, the configuration complexity we observe in dynamically changing infrastructure clouds calls for tool-support. Existing research in this space is mostly focused on dynamic infrastructure analysis of non-security properties [14], node integrity monitoring [13] or establishing security analyses of static systems given by a configuration snapshot [2]. While the latter results give us confidence about reasoning on security consequences of infrastructure cloud topology and configurations, they suffer from blind spots due to transient security failures as well as from efficiency problems. In fact, an isolation case-study [3] using this approach showed that the analysis of a mid-sized virtualized infrastructure required about *seven minutes* for extracting the configuration and building up a model and *one minute* on the actual analysis of the model. Performing such an analysis in a dynamic environment will lead to a backlog of changes that need to be analysed, an increase in the response times in case of security incidents, as well as scalability and efficiency problems.

Consequently, it is our primary goal to reduce the times for configuration extraction, model building and analysis by establishing a systematic *differential* approach that does not require a full extraction and analysis on each configuration change, but still maintains strong security foundations all the way. We realise this goal with a practical security system that uses a model-based security analysis. It maintains a graph representation synchronised with the actual configuration of the virtualized infrastructure and accepts change events produced by cloud management hosts to update its own model. The model and its updates form the foundation for a differential security analysis that maintains an information flow graph for analysing isolation properties and which tries to find violations of specified security policies.

**Our Contributions.** With the overall research goal to establish a differential security analysis of dynamic infrastructure clouds, we make the following contributions: **1)** We establish an architecture that caters for near to real-time detection of configuration changes in heterogeneous virtualized infrastructures. **2)** In order to maintain a synchronised graph model of these infrastructures, we propose a set of algo-


*ACSAC'14,* December 08–12, 2014, New Orleans, LA, USA.

ACM 978-1-4503-3005-3/14/12 ...$15.00.
http://dx.doi.org/10.1145/2664243.2664274.

rithms for the computation of graph deltas (added/removed nodes and edges, changed node attributes) applicable to a graph model based on change events. **3)** We propose a novel approach that soundly maintains an information flow graph for the dynamic graph model of the infrastructure. **4)** We offer a practical implementation of our system, called *Cloud Radar* (*CR*), for VMware environments. Our comprehensive evaluation shows that the differential approach reduces the overall analysis time significantly, putting near-to-real-time analysis in our reach. For a broad spectrum of cloud operations and even for large infrastructures, we measure model update times in the order of milliseconds, which renders our approach several orders of magnitude more efficient than previous static analysis approaches. **5)** We establish a security analysis showing that *Cloud Radar* can be set up as security monitoring of insider adversaries.

# 2. SYSTEM AND SECURITY MODEL

An infrastructure cloud consists of (virtualized) computing, networking and storage resources, which are configured through a management host and its well-defined interface.

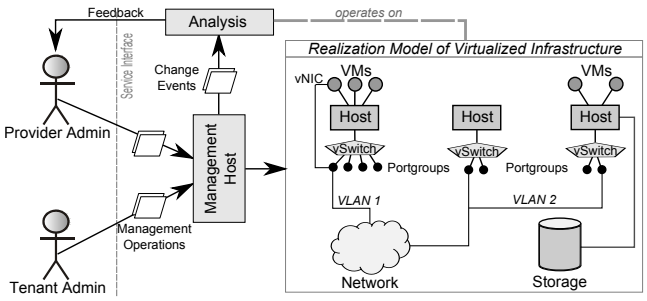



**Figure 1: System model of the differential security monitoring covering compute, network, and storage.**

As shown in Figure 1, the system model of this work is poised towards a differential analysis based on change events issued by the management hosts when the infrastructure is re-configured. The analysis system uses these change events to continuously update a graph representation of the infrastructure, the *Realization* model, which is used for subsequent analysis. As long as the management host issues the events correctly, the model covers malicious adversaries, insiders and externals alike.

## 2.1 System Model

We represent the virtualized infrastructure in a graph model, called *Realization* model (cf. [3]): The model is an undirected, vertex typed and attributed graph. The vertices of the graph represent the components of the virtualized infrastructure, which may be entire sub-systems, such as physical servers or virtual machines, or low-level components, such as virtualized network interfaces. Vertices are typed, e.g., type vm denotes a virtual machine, and annotated with name/value attributes. The attributes encode detailed properties of the components and capture their configuration. The edges of the graph represent the connections and relationships among components of the virtualized infrastructure, therefore encoding its topology. The vertex types of our model are organised in a hierarchy graph, i.e., a directed acyclic graph (DAG) where the edges represent a parent-to-child relation. The hierarchy graph reflects the inherent hierarchy found in the infrastructure. For example, a virtual machine belongs to a physical host, and therefore a physical host has a directed edge to a virtual machine.

Considering the example from Figure 1, we see that the Realization model captures all areas of the virtualized infrastructure: computing, networking and storage. While the actual model encodes fine-grained components of all these areas, e.g., storage being represented as virtual disks, file backend objects and storage pools, we focus our explanation on the networking components to prepare the ground for examples in subsequent sections. Physical hosts and their hypervisors provide networking to VMs by virtual switches that connect 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 by assigning distinct VLAN IDs to the associated port groups.

## 2.2 Threat Model

We establish a threat model based on the dependability taxonomy [1]. Users and administrators can be malicious or non-malicious. Thereby, we cover all classes of human-made faults, independent from *intent* or *capability*, that is, faults can be introduced deliberately as result of a harmful decision or without awareness; faults can be introduced by accident or by incompetence. These fault classes include misconfigurations as well as malicious insider administrators and, thereby, constitute a strong adversary model. Agents that operate on behalf of a human, e.g., as part of cloud automation, are also covered by this threat model, because we do not differentiate between the issuers of changes.

We only place one constraint on how the adversary can exert threats upon the virtualized infrastructure: The adversary is bound to the well-defined cloud manager API and cannot subvert the communication channel between the management hosts and the analysis system. In §5 we discuss and assess multiple deployment approaches to realise such a constraint in a practical environment. For example, based on isolation of the monitoring and management networks from the administrators, as well as using mandatory access control. Note that we consider the security of the software for the management host and the hypervisors as out of scope.

# 3. DESIGN AND IMPLEMENTATION

In this section, we describe the design and implementation of our *Cloud Radar* system. The goal of our system is to detect – in near real-time – configuration changes that impact the security of virtualized infrastructures. On a high-level, the system works in the following way.

The start of the system's workflow is an initial snapshot of the configuration and topology of the entire virtualized infrastructure represented as a graph model. An initial information flow analysis determines how information may flow within the infrastructure, in order to determine isolation properties. Isolation is critical in multi-tenant virtualized infrastructure and the concern of many security policies. The crucial part of our approach is that we operate on *change events*, which are the result of a change in the infrastructure, and which needs to be represented in the model by transforming it according to the event. The transformation of the model may result in new or changed information flow in the system, and the information flow analysis is differentially up-

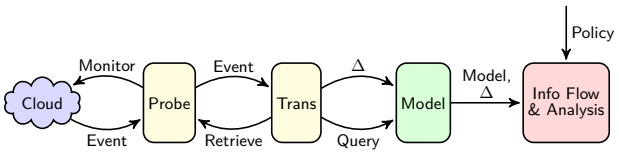



**Figure 2: System Architecture and Components.**

dated based on the change event. Finally, after each change the resulting model will be analysed with regard to a given security policy or a set of policies.

We depict the system architecture and components that implement such an analysis workflow in Figure 2. The system is composed of the following components. A **Probe** knows how to monitor the virtualized infrastructure and how to obtain change events. It is tied to a specific virtualization technology, e.g. VMware. For heterogeneous environments multiple probes are instantiated. For each probe a translation component (**Trans**) exists that knows how to convert from change events into model transformations, i.e., a graph delta ($\Delta$). Depending on the information richness of the events, further information may need to be retrieved from either the probe or queried from the existing model.

The **Model** component contains the graph model of the virtualized infrastructure as introduced in §2.1. By obtaining graph deltas ($\Delta$) from the *Trans* component, the model is updated in accordance to the change event. A graph delta contains the nodes and edges that should be added or removed, as well as attribute changes for nodes. As a pre-processor for the analysis, the **Info Flow** component determines the information flow implications in the infrastructure and updates the graph model with information flow edges. In a differential analysis, the information flow is updated based on the graph delta. The **Analysis** component analyses the infrastructure given as the graph model with regard to a security policy, which expresses desired or undesired properties of the infrastructure topology or configuration.

## 3.1 Obtaining Infrastructure Change Events

We follow a similar architecture as presented in [3] where a set of *probes* extracts the configuration of different virtualized systems. Instead of periodically extracting the entire configuration, we extend and improve the existing approach with probes that obtain events of changes in the infrastructure.

The format and level of information of change events can largely vary between different virtualization technologies. For example, VMware and Xen provide rich events on changes in their inventories, Libvirt provides change events on a VM level, and OpenStack as a management platform only provides coarse-grained events. In our design, we cater for the variety of formats and information richness of events among different virtualization technologies. In this paper, we focus on the VMware event probe.

### 3.1.1 VMware Probe

VMware maintains an internal relational inventory of the virtualized infrastructure that is composed out of *Managed Entities*, such as virtual machines or physical hosts. Managed entities can have properties that describe further configuration aspects of that entity. Each entity can be addressed using a *Managed Object Reference (MOR)*.

We are using the method WaitForUpdates of the VMware API [16] to obtain notifications on updates and property changes for managed entities. This method is part of the *Property Collector* component, which also handles retrieval of properties of entities in the API. The method returns an UpdateSet object that contains an incremental version number, which is used in repeated calls to only obtain the latest changes. Further, the update contains a set of ObjectUpdate objects with an *Object* attribute stating the *MOR* of the updated object, as well as a *Kind* attribute to indicate the kind of update. The update kind can be i) *Enter* for a new object, ii) *Leave* for a removed object, and iii) *Modify* for property changes of that object.

Essentially, the updates state which objects have been added or removed from the inventory, and which have been modified. For new or modified objects, a set of PropertyChanges describe the property changes of the objects in the following form. **Operation:** The type of property change. It can be i) *Add* a value to a collection property, ii) *Remove* a value from a collection, or iii) *Assign* a value to the property. **Name:** The name of the property that is changed. **Value:** Only for Add and Assign, the value that is added or assigned to the property.

Consider the operation UpdatePortGroup that allows an administrator to change the virtual networking configuration for virtual machines, including changes to the network isolation property. In the case that an administrator changes the VLAN ID associated with a port group *PG-1* to a new value of *123*, we obtain the following event from VMware.

```
[modify] HostSystem (host-159)
  assign: config.network.portgroup["key-vim.host.↩
      PortGroup-PG-1"].spec.vlanId <- 123
```

The event indicates that the host object host-159 has been modified. In particular, as part of the network configuration of that host, the property spec.vlanId of the port group *PG-1* has been assigned the value *123*.

## 3.2 From Change Events to Model Updates

From a high-level perspective, the *Trans* component translates from a change event to a model update in the form of a graph delta, as illustrated in Figure 2. The translation has to differentiate between three kind of change events. First, a new object appeared that may result in new nodes and edges in the graph model. Second, an object was removed and the corresponding nodes and edges in the model have to be removed, too. Finally, an object has been modified, i.e., attributes of that object have changed. This may result in attribute changes of nodes in the model too, but it can also leads to the creation or deletion of nodes and/or edges in the model. This categorisation aligns well with the events produced by the VMware probe since they contain an attribute indicating the kind of change.

A translation is typically bound to a specific probe and its produced event format, i.e., a VMware translation knows how to translate VMware change events. Therefore, we focus in the following on describing the event translation design with the concrete example of translating VMware change events. The translation needs to handle the three different change events, but also has to deal with the ordering of events due to their dependencies, and with incomplete new objects. Therefore, the output of the translation is either a graph delta, dependency requirements, or a notification that the translation encountered an incomplete, ignored, or unsupported object. If an change event consists of multiple object updates, we merge the produced graph deltas to form a single graph delta for that change event.

### 3.2.1 Translation of an Object Update

We explain and propose a set of algorithms for the successful translation of an Object Update into a graph delta. The handling of a failed translation due to cases such as incomplete objects or dependency ordering will be discussed in §3.2.3 and §3.2.4 respectively.

We define a graph delta as $\Delta = (V^+, V^-, E^+, E^-, M)$, where we differentiate between *creator* nodes and edges $(V^+, E^+)$, *eraser* nodes and edges $(V^-, E^-)$, and a set $M$ of node attribute modifier in the form of $(node, attribute, value)$. Creators lead to new elements in the graph, erasers remove existing elements from the graph, and node attribute modifiers change attributes of existing nodes to new values.

#### *Enter Object: Creating New Nodes and Edges.*

We obtain an *Enter* Object Update for a new object that has been created in the inventory as well as for all the existing objects in the inventory during the initial probe connection. The goal of the translation component is to produce new nodes and edges for the model based on the update.

The fundamental idea is to employ a recursive algorithm that starts at a newly created object and traverses through all its connected neighbour objects. For each object, the technology-specific parts of the translation creates a corresponding model node and populates its attributes with values of the object. It further establishes relations to other created nodes due to the recursion. The output of this algorithm is a set of newly created model nodes and edges, where the edges not only connect to new nodes, but also to existing nodes in the model.

The translation of object updates has to handle two corner cases: Incomplete objects, where the attributes of an object have not been populated fully yet, and the ordering of object updates within the same update set. We will describe the handling of these cases in §3.2.3 and §3.2.4, respectively.

#### *Leave Object: Deleting Nodes and Edges.*

For each object that has been removed from the inventory, we obtain an *Leave* Object Update. The update contains the MOR of the removed object, and we lookup the corresponding model node identifier and obtain the node by querying the model component. Since in our model a managed entity might have resulted in the creation of multiple nodes, we have to perform a recursive deletion of the dependent nodes of the removed node.

The recursive deletion works as the following. First, given an object that was removed from the inventory, we lookup the corresponding node in the model, and add the node to the eraser node set. For all the deleted node's neighbours, we place the connecting edges in the eraser edge set. Further, we continue the recursive deletion at the neighbour node if i) the neighbour's type is a child type in the hierarchy (cf. §2.1); and ii) the neighbour node is not a managed entity.

#### *Modify Object: Creating an Entire Graph Delta.*

Finally, we consider the case that an object has been modified. A *Modify* Object Update consists of a Property Change and the modified object reference. This property change indicates the type of change, the attribute that has changed, and potentially a new value.

Our algorithm consumes such a property change and produces a graph delta that consists of creator/eraser nodes and edges, as well as a set of attribute modifier in the form of $(object, attribute, value)$. The algorithm has a similar structure as the algorithms for creating or removing objects, and in fact builds upon them. For each object type, we further differentiate between the changed attribute, as well as the operation performed on that attribute. For attribute assignments, we construct attribute modifiers that change the corresponding node's attribute. For added managed entities or data objects, we rely on the algorithm that handles *Enter* objects. Similarly, we construct erasers for deleted objects based on the *Leave* object algorithm.

An example to illustrate a modified object is the creation of a new virtual device, such as a virtual ethernet adapter, for a VM. In this case, we have to translate the new virtual device into a new model node, and connect it to the existing VM node with an edge. Such an event produces the following creator nodes and edges: $V^+ = \{vnic, vport\}$ and $E^+ = \{(vm, vnic), (vnic, vport), (pg, vport)\}$, where $vm$ corresponds to the existing VM, $vnic$ and $vport$ are created, and the virtual port is connected to the port group $pg$.

### 3.2.2 Applying the Model Update

Given a graph delta as produced by our set of algorithms based on a change event, updating the graph model is expressed as updating the node and edge sets of a given graph $G = (V, E)$: $V' = (V \setminus V^-) \cup V^+$ and $E' = (E \setminus E^-) \cup E^+$. The updated model graph is $G' = (V', E')$. For each node attribute modifier $(node, attribute, value)$ in $M$ we change the attribute of the node in $V'$ with the new value.

### 3.2.3 Postponing Incomplete Objects

Resolving further information of an object during the translation may fail when not all relevant attributes have been set yet. For example in VMware, when a VM is created its configuration is only later fully populated. In that case, we are dealing with an incomplete object. We maintain a set of incomplete objects and monitor updates for these objects. If an incomplete object receives an update, we try to handle it as a new object rather than a modified one. If the translation succeeds, i.e., the object was complete and could be translated, we remove the object from the incomplete set. Otherwise, it remains in the incomplete object set. In practice and during our evaluation, incomplete objects always received modify events when further attributes were populated, usually within a sub-second time span. We may also employ periodic translation attempts for incomplete objects, in case they receive no further modify events.

### 3.2.4 Ordering of Updates based on Dependencies

We are dealing with an asynchronous system and we have to take care about the ordering of the change events. However, two aspects of the VMware probe supports the ordering of events. First, the probe connects over TCP which provides packet ordering for us. Second, VMware employs version numbers for the event discovery, which provides an ordering of events between different versions. However, within one UpdateSet, we may encounter a wrong ordering of changes. We can order the changes by using a dependency graph, i.e., a directed acyclic graph (DAG) where vertices are changes and directed edges describe dependencies such that the source node fulfils the dependency of the target node.

We construct such a dependency graph in the following way. A successful translation of an event returns a graph

**Table 1: Subset of Information Flow Rules Relevant for Portgroup VLAN Isolation.**

| # | Type | Flow | Node Pair | Condition(s) | Edge Dependency |
|---|---|---|---|---|---|
| 1 | Simple | noflow | $VSwitch \Leftrightarrow PortGroup$ | Portgroup's VLAN ID != 0 | Attribute VLAN ID |
| 2 | Complex | flow | $PortGroup \Leftrightarrow PortGroup$ | Portgroups' VLAN IDs equal and their virtual switches connected | Attribute VLAN ID, Connectivity of VSwitches |
| 3 | Simple | flow | $Any \Leftrightarrow Any$ | None | |

delta of new or modified nodes. In the case of an unsuccessful translation due to a missing dependency, i.e., a node was not found in the current model, the translation returns a *requirement* in the form of a node type and a predicate on its attributes. Further, the translation of an event may return *potential* nodes, which become available once other requirements are fulfilled. Based on the translation attempts, we try to match new or modified nodes with requirements, and introduce a directed dependency edge from the fulfilling event to the requirement. Potential nodes may also satisfy requirements, thereby building up a dependency graph. A topological sort of the dependency graph will yield an evaluation order.

## 3.3 Differential Information Flow Analysis

The information flow analysis determines how information may flow in the virtualized infrastructure by computing an information flow graph. The graph forms the foundation for analysing isolation properties in the infrastructure. The analysis works in two phases: first, it takes the realization model graph that represents the infrastructure and computes an overlay directed information flow graph. Second, it computes for the information flow graph the strongly connected components (SCC), i.e., the sets of graph nodes that are mutually reachable, and constructs a reachability graph of the SCCs. Our analysis is inspired by [3], but has been extended and improved to be more efficient and operate in a differential way, i.e., it operates on changes of the realization model rather than computing the information flow from scratch after each change. In order to compute the SCCs, we employ Tarjan's algorithm [15], but variants also exists that operate in a differential way [12].

### 3.3.1 Specifying Flow and Trust Assumptions

The core of the information flow analysis is a set of *traversal rules* that specify which elements in the infrastructure are trusted and provide isolation, and which elements constitute to information flows. For example in VMware, port groups provide isolation if they have been configured with a VLAN ID different than zero. The traversal rules for this example are listed in Table 1. A first-matching application of traversal rules results in new information flow edges, which either describe a *flow* or *noflow* (isolation) between a pair of nodes. Further, an information flow edge can be dependent on a specific node's attribute, the connectivity of nodes, as well as a combination of both. We differentiate between *simple* rules, which determine flows between an adjacent pair of nodes, and *complex* rules, which work on a pair of nodes that are not necessarily adjacent, but that fulfil a common condition such as equality of an attribute.

In our example, a simple rule introduces a *noflow* edge between a virtual switch and a adjacent port group in case the VLAN ID is not zero. The edge is attribute-dependent on the port group's VLAN ID attribute. Further, a complex rule introduces *flow* edges between non-adjacent port groups

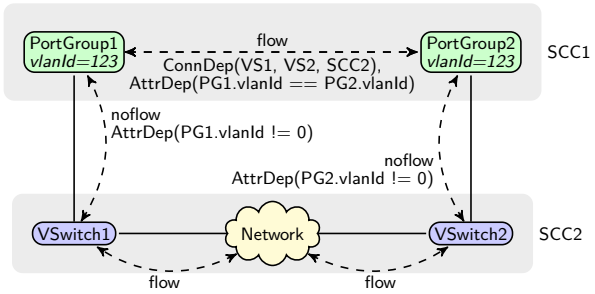



**Figure 3: Graph model annotated with dashed information flow edges of different kinds (simple, attribute-dependent, connectivity dependent).**

of the same VLAN ID (attribute dependency) if the virtual switches of the port groups are connected (connectivity dependency). A default rule introduces flow between any node pair that has not been covered by a previous traversal rule. Figure 3 illustrates the resulting topology graph that is annotated with information flow edges, as well as grouping nodes into SCCs, i.e., SCC1 and SCC2. In the case of the connectivity-dependent flow edge between the port groups, we see that the virtual switches are recorded as *connectivity endpoints* and SCC2 is part of the *connectivity path*.

### 3.3.2 Maintaining an Information Flow Graph

The challenge we solve is to maintain an information flow graph, which is build from simple as well as attribute and/or connectivity-dependent information flow edges, even when connectivity or attributes change. Our differential analysis works in two phases: 1) Given a graph delta (cf. §3.2.1), we compute the insertion and deletion of nodes and edges from the information flow graph by identifying affected graph elements and selectively apply traversal rules; 2) Based on the previous insertion and deletion of information flow edges we compute the reachability graph of strongly connected components. In case of changes in the connectivity, we also introduce or remove connectivity-dependent edges.

#### *Processing Realization Model Graph Deltas.*

In the first phase we process the realization model graph delta, and for each of the following elements of the graph delta compute information flow graph changes.

*Creator Nodes:* We insert the new nodes in the information flow graph. For each new node, evaluate the complex traversal rules, which may create new information flow edges, and insert the created edges in the information flow graph.

*Creator Edges*: We evaluate the simple rules for each node pair of the new edges and insert the resulting information flow edges in the graph.

*Eraser Nodes:* We remove the nodes as well as all their incoming and outgoing edges from the information flow graph. Further, we find all connectivity-dependent edges that require connectivity of a removed node, and remove those edges too. For example if VSwitch is removed from Fig. 3, the edges to

PortGroup1 and Network are removed. Additionally, the edge between the port groups is removed, because it is dependent on the connectivity of the vswitches.

*Eraser Edges:* For each undirected removed realization model edge, we find the directed information flow edges and remove them from the information flow graph.

*Node Attribute Changes:* Finally, we find all affected attribute-dependent edges and remove them if their attribute condition does not hold anymore. The simple rules are re-evaluated for the invalid edges and the complex rules are evaluated for the changed nodes. The resulting information flow edges are inserted in the graph. For example, if PortGroup1's VLAN ID changes to zero, the edges between the vswitch as well as the other port group are removed, but a new flow edge is introduced between the vswitch.

### *Processing Connectivity Changes.*

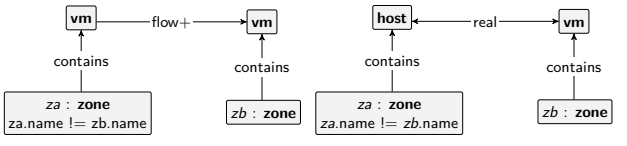
In the second phase of our analysis, we compute a new SCC reachability graph when information flow edges have been inserted or deleted in the first phase. We detect changes in connectivity by comparing the previous with the new SCC reachability graph. In particular, we operate on new and removed SCCs as well as inter-SCC edges, and either create or remove connectivity-dependent information flow edges.

*Removed SCCs or inter-SCC edges:* In the case of *reduced* connectivity, we find all connectivity edges that contain a removed SCC or removed inter-SCC edge in their connectivity path, and remove such edges. For example, if the Network node is removed from the example of Fig. 3, SCC2 splits into two new SCCs, i.e., SCC2 is removed and two new SCCs are added. In the example, the connectivity-dependent edge between the port groups is affected and removed, because SCC2 appears in its connectivity path. Since another connectivity path could exists for the connectivity endpoints, we re-evaluate the complex rules for the removed edges' node pairs. In the example no other connectivity path exists.

*New SCCs or inter-SCC edges:* In the case of *increased* connectivity, we re-evaluate connectivity candidates, i.e., information flow edges that previously have been missing connectivity, if they are affected by the new SCCs or new inter-SCC edges. If their connectivity is now fulfilled, we add them to the information flow graph.

## 3.4 Specification of Security Policies and Detection of Policy Violations

For the detection of security failures, we define the following security policies, in the form of *attack states*, with their graphical representation shown in Figure 4. *Cloud Radar* tries to match the policies on the dynamic realization model and information flow graph. Once a policy's attack state matches, we have found a security failure. A set of security administrators is notified about a security violation, in order to mitigate the problem.




**(a) Network Isolation** **(b) VM Placement**

**Figure 4: Graphical Representation of Network and Compute Security Policies.**

*Network Isolation:* Virtual machines are grouped into "security zones", e.g., production and test zone, and these zones must be isolated on the network level, e.g., through different virtual networks. This policy is violated if we find a potential connection (flow+) between two VMs of different security zones (za and zb).

*VM Placement:* A group of virtual machines should run on one or multiple designated physical hosts, e.g., for performance, availability, or also data privacy reasons. This policy is violated if a VM runs on a different host than the ones designated. Preventing VM co-location, e.g., due to side-channel attacks [11], is a variant of this policy.

*Storage Isolation:* VMs of different security zones must not be able to exchange information over a shared storage device, e.g., by using the same file as backing of the VMs' virtual disks.

We have two implementations to find a policy violation by matching the policy's attack state against the current realization model and information flow graph. The first one is a native implementation in Java/Scala that iterates through the nodes in the model graph. It benefits from a fast execution time and uses the SCC reachability graph to efficiently determine if two model nodes are connected. Either the two nodes are in the same SCC or there exists a path between their corresponding SCCs in the reachability graph. Otherwise, they are not connected. However, implementing new policies requires a native implementation, which makes it less extensible by end-users, such as a cloud administrator.

The second implementation uses a general-purpose graph matching tool called Groove [6], which tries to match a given sub-graph in a larger graph. In fact the policies in Fig. 4 are valid sub-graphs that Groove can match against our realization model graph. The main benefit of this approach is its extensibility, since end-users can implement new policies in a graphical and intuitive way. However, as a general purpose tool it bears a higher execution overhead and for determining connectivity it uses an equivalent but less efficient path-finding algorithm, compared to the SCC approach.

# 4. PERFORMANCE EVALUATION

In this section we empirically evaluate and discuss the performance of *Cloud Radar* in the case-study of a semi-production environment as well as in simulated environments of different sizes. The performance evaluation focuses on the processes of building and maintaining the models in sync with changes in the infrastructure.

## 4.1 Methodology and Environments

Our evaluation is performed with different environments: a real, semi-production environment ($R_{150}$) with 2 hosts and 150 VMs, and a simulated environment that uses an infrastructure simulator incorporated in the VMware management hosts. We vary the size of the simulated environment ($S_{\#VMs}$) between 150 and 30,000 VMs with a host-VM ratio of 1 : 50. This allows us to evaluate the scalability of our approach. *Cloud Radar* itself runs in a Linux VM with 12 vCPUs, 12 GB RAM, and Java 1.7.

We differentiate between the two approaches of obtaining and maintaining the model of a virtualized infrastructure: *Static Snapshot* is the existing approach that always extracts the full configuration. In order to deal with a dynamic and

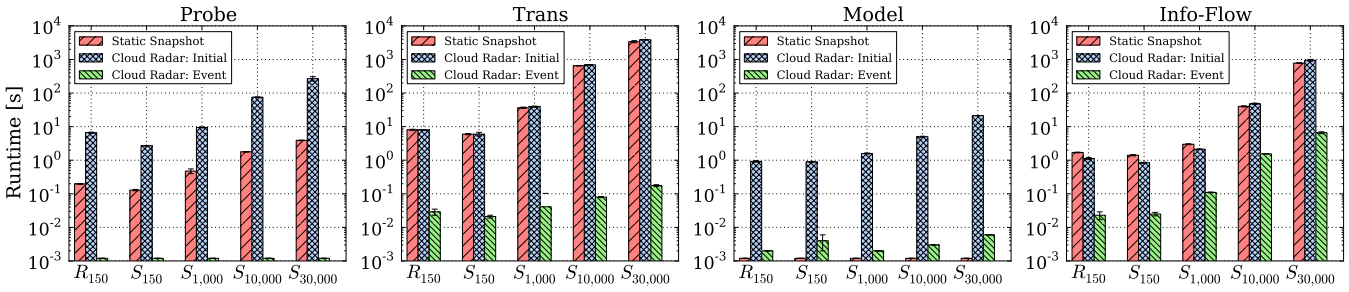



**Figure 5: Runtime measurements (in seconds, log scale) of the existing approach "*Static Snapshot*" and the new *Cloud Radar* approach ("*Init*" for initialisation and "*Event*" for change events) for the four system components in relation to the infrastructure size (number of VMs).**

constantly changing infrastructure, the extraction has to be executed periodically. On the other hand, *Cloud Radar* obtains an initial event containing the full configuration of the infrastructure, followed by events for infrastructure changes. In order to compare the performance of the two different modes, we measure the runtime of the Probe, the time needed to translate the Probe output into a model (sub)-graph, to initialise or update the graph model, as well as to construct or maintain the info flow graph. For the new event-based approach, we measure these aspects for both the initial event and subsequent change events. In order to trigger change events, we automatically perform a variety of operations on the virtualized infrastructure. In the particular measurement of Fig. 5 we used the CreateVM operation.

## 4.2 Results and Discussion

Figure 5 illustrates the main results of our performance evaluation with the runtime in seconds on the logarithmic y-axis, and the different environments and sizes in terms of number of VMs on the x-axis. We break down the results into measurements for the four system components. The measurements for the existing approach are shown as the *Static Snapshot* bars, and the new approach is broken down into measurements for initialisation (*Cloud Radar: Init*) and events (*Cloud Radar: Event*). Our measurement resolution is $1ms$ and measurements such as the runtime of the probe for events is equal or below that resolution.

*How does the new event-based approach compare to the existing full extraction approach? How does the system scale with the size of the virtualized infrastructure?*

First of all, comparing the results of the 150 VM sized realistic ($R_{150}$) and simulated ($S_{150}$) environments show equivalent results, which indicates that the infrastructure simulator in fact behaves accurately and provides a suitable environment for performing our measurements. Of course, in a realistic environment our absolute measurements could differ, but the scalability of the system would remain equivalent.

**Probe:** We observe that both approaches initially scale linearly, although the runtime of the existing approach is lower compared to the initialisation phase of our new one. We suspect this behaviour to be rooted in the more complex construction of the probe output. The existing approach traverses the VMware inventory and obtains a list of all the managed entities. In contrary, the new approach sets up a filter and VMware is required to find all entities that match the filter and have not been reported previously. However, after the costly initialisation, the probe reports events instantly and below our measurement resolution of $1ms$, independently of the infrastructure size.

**Trans:** This is the dominating factor in the initialisation of the model, in particular for large-scale infrastructures. Both the existing approach as well as our new approach perform almost identical for initialising the model, and scale linearly with the size of the infrastructure. This is unsurprisingly, as both the existing approach as well as the new approach with *Enter* events perform a similar translation of creating new objects. The significant performance improvements lies in translating change events into updates of the model with our new approach. To highlight this, consider the $S_{30,000}$ environment with 30,000 VMs: After the comparable initialisation time by both approaches, the existing approach would require a periodic translation of the entire environment taking *56 minutes*, whereas in the new approach each change event can be translated in *176 milliseconds* (the worst-case we measured for our set of operations). This is an improvement of four orders of magnitude.

**Model:** Both the model initialisation of the existing approach as well as the model update based on events are almost instantly. In the first case, a new full model is constructed all the time and can override the existing one, i.e., a simple reference assignment. In the latter case for the operations we tested, the graph delta remains small and is merged into the existing graph model. For the initial large event in our new approach, we have to merge a set of graph deltas together, where the size depends on the infrastructure size, resulting in a linear scalability. This also indicates the worst-case scenario, in case an operation results in an event that changes the entire infrastructure.

**Info Flow:** If we break down the information flow analysis (cf. Table 2), we observe for the full analysis a linear scalable evaluation of simple traversal rules and SCC computation. We also see a quadratic complexity for evaluating the complex traversal rule, which needs to evaluate pairs of port groups in our example rule set (cf. Table 1). The differential approach provides significant improvements with a rules evaluation that only depends on the size of the event and a linear SCC computation, for which we see even further potential for optimisation.

**Analysis:** We measured a runtime of $19ms$ for finding violations of the network isolation policy in the real environment. This includes finding all violations of the policy, although one could terminate after the first violation. The VMware infrastructure simulator does not support operations that trigger such policy violation, therefore our performance measurement is limited to the real environment.

In summary and in the light of the more expensive initialisation of the new approach, when does it actually pay off? Consider the 10,000 VM environment $S_{10,000}$ and the

**Table 2: Breakdown of Info Flow Runtime (in $ms$) into Simple/Complex Traversal Rule and SCC Computation for Static- and Event-based Approaches.**

| | Simple | | Complex | | SCC | |
|---|---|---|---|---|---|---|
| | *Static* | *Event* | *Static* | *Event* | *Static* | *Event* |
| $R_{150}$ | 153 | 4 | 151 | 1 | 153 | 48 |
| $S_{150}$ | 157 | 4 | 90 | 2 | 139 | 52 |
| $S_{1,000}$ | 269 | 4 | 592 | 2 | 465 | 138 |
| $S_{10,000}$ | 755 | 6 | 31,172 | 2 | 2,776 | 1,095 |
| $S_{30,000}$ | 1,681 | 4 | 926,708 | 2 | 11,346 | 4,769 |

cumulative runtimes of both approaches. The initialisation in the existing approach overall takes $693s \pm 14$ and for the new approach $819s \pm 20$. While follow up model updates require a full periodic execution of the entire workflow in the existing approach, the new one only requires $1.8s$ for each change event. Although the new approach is slightly more expensive in the initialisation, even after two event it pays off due to the much more efficient event processing.

*How many events can be processed per second until we run into a backlog?* Considering the simulated 10,000 VM environment ($S_{10,000}$), we can observe and translate approximately *33* VM creation operations per minute, bounded by the dominating translation time of $1.8s$ per CreateVM operation and assuming a serialised processing.

# 5. SECURITY EVALUATION

We evaluate the security of *Cloud Radar* in two ways. First, a security analysis argues that all change events are received with integrity, in face of the given adversary model. Further, we discuss various approaches how our system can be deployed securely in practice. Second, we test the system's ability to detect policy violations for compute, network, and storage resources using randomised operations.

## 5.1 Security Analysis

The security analysis considers the management host creating events and the *CR* host as separate entities and considers multiple attack vectors including manipulating network communication through VLAN re-configuration or denial of service or dropping communication sessions in the Session Manager. We propose a practical deployment of *Cloud Radar* that is secure in the face of an insider adversary, based on a small set of assumptions and a deployment pattern, which includes isolation of the reporting network, a heartbeat signal, and mandatory access control for regular administrators.

### 5.1.1 Assumptions and Deployment Pattern

The threat model of §2 already introduces that software attacks are out of scope, which includes that the management host software cannot be manipulated by the adversary. Our analysis is built on the following explicit assumptions, which form the backdrop of the deployment pattern.

[secchan] TLS offers a secure channel providing channel confidentiality and integrity, with server authentication based on a dedicated PKI. The adversary does not have capabilities to establish host certificates in the certificate tree of the root CA $\mathsf{CA}_{\mathsf{sec}}$ trusted by CR.

[access] The adversary accesses the virtualized infrastructure through the management interface only. This implies that the adversary does neither have physical or root access on the physical hosts, direct access to the hypervisor nor physical access to network and storage. The adversary does not have access as super_admin.

The assumption [access] is motivated by vSphere Security [17] best practice, which states that hypervisor hosts should only be managed through the central management host. This can also be enforced by putting the hypervisor into *lockdown mode*, by which no other users than vpxuser, the vCenter management user, have authentication privileges nor can perform operations on the host directly.

**Network Isolation:** We need to establish the condition [netisolation] that the reporting network (between the management host and Cloud Radar) is isolated from the networks accessible by the adversary to protect the event channel from interference. A dedicated reporting network $\mathsf{net}_{\mathsf{sec}}$ is created for the event reporting between management host and *Cloud Radar*. The network isolation is enforced 1) as dedicated physical networks (building upon the assumption [access]), 2) with a VLAN in the physical switch, where hypervisor or virtualization administrators do not have privileges, or 3) as a virtual network with a dedicated VLAN ID, where the administrators do not have privileges to change the VLAN configuration. The event channel is established as a secure channel ([secchan]) to the management host via $\mathsf{net}_{\mathsf{sec}}$.

**Heartbeat Signal:** The condition [heartbeat] models the realisation of a heartbeat signal sent in time intervals $t_{\mathsf{hb}}$. A heartbeat can be realised by 1) opening the CR probe filter to background noise events, such as machine utilisation, including them into the event stream, 2) a periodic task changing managed entities scheduled by the super_admin, or 3) CR exercising write access on the managed entities, e.g., VMs, to obtain change events directly. It is necessary that the heartbeat signal will be in the event channel observed by the CR probe. Whereas the first approach is least invasive and does not require write privileges, it may yield false positives. The two other approaches give a reliable heartbeat signal, yet require partial write access, either under control of the super_admin or *Cloud Radar* itself.

**Mandatory Access Control:** The super_admin sets privileges such that regular administrators only gain privileges on the management host, but not on the hypervisors according to [access]. The following privileges are set on the management host: **1)** No administrator has rights to revoke a lockdown mode of a host. **2)** No administrator has rights to manipulate $\mathsf{net}_{\mathsf{sec}}$. **3)** The administrator privileges for session manipulation on Sessions are restricted, in particular Sessions.TerminateSession is controlled.

### 5.1.2 Security Argument

The foundation of *Cloud Radar* (*CR*) to detect security failures is the ability of obtaining all change events of the infrastructure in an unmodified form. Therefore, we establish the requirements integrity and availability, and argue that our secure deployment of *CR* fulfils these requirements.

*Integrity.*

For any $n$-th event $e_n$ received at *CR* holds that the event is correct, fresh, in order and as it has been sent by the management host.

According to §3.1.1, if *CR* requests version $n$, the management host is guaranteed to produce an event $e_n$, which contains all changes after $e_{n-1}$ up to reception of the request for version number $n$. Thereby, the event chain is complete. We obtain the order and weak freshness properties from the

version number, as an event $e_n$ must have been generated after any event $e_{<n}$.

The event $e_n$ is received at *CR* over a secure channel according to condition [netisolation]. The channel is established over the dedicated network $\mathsf{net}_{\mathsf{sec}}$ and server-authenticated on cert that is in the certificate chain of trusted $\mathsf{CA}_{\mathsf{sec}}$, which is inaccessible to the adversary according to [secchan]. Thereby, the connection is with the correct management host. Further, based on the secure channel of [secchan], we obtain channel confidentiality and integrity on the event $e_n$, which is thereby as sent by the management host. As the adversary can neither interfere with the management host event reporting by the exclusion of software attacks nor with the network configuration for $\mathsf{net}_{\mathsf{sec}}$, the event $e_n$ is the correct event sent as intended by the management host. From [access], we obtain that the adversary could not have changed the event at the management host or any subordinate host.

*Weak Availability.*

Either all events sent by the management host are received by *CR* eventually and latest within a channel timeout $t_{\mathsf{timeout}}$ or an alarm is raised after $t_{\mathsf{timeout}}$ is elapsed.

The network $\mathsf{net}_{\mathsf{sec}}$ is modelled as an asynchronous channel, through which messages arrive eventually, the secure channel is established over it by *CR*. Observe that even though underlying TCP/IP offers reliable, ordered and error-corrected communication, it does not give strong timeliness guarantees. Whereas the secure channel enforces integrity and ordering, it does not offer availability. Because of the in-order delivery of $\mathsf{net}_{\mathsf{sec}}$, it follows that if $e_n$ is received, then all previous events $e_{<n}$ must have been received already, yielding that, if the channel is intact, all events sent by the management host are received by *CR* eventually and latest within a set timeout $t_{\mathsf{timeout}}$. The condition [netisolation] isolates the network $\mathsf{net}_{\mathsf{sec}}$ from interference by the adversary on the network, while [access] prevents interference on the subordinate hosts, however this does not rule out availability failures from other root sources, e.g., a cable fault.

The Weak Availability clause, i.e., an alarm is raised after $t_{\mathsf{timeout}}$ is elapsed, is obtained from the condition [heartbeat]. If the channel waits for a packet or the channel is interrupted, then we have that eventually $t_{\mathsf{timeout}}$ will be reached without a packet having arrived at *CR*. According to [heartbeat], the management host produces a heartbeat signal after each time window $t_{\mathsf{hb}} < t_{\mathsf{timeout}}$. Therefore, we have that the after $t_{\mathsf{timeout}}$ without a message, *CR* can conclude that the channel is interrupted and raise an alarm. It follows that availability failures are detected within $t_{\mathsf{timeout}}$. The system will try to reestablish the connection after an interrupt.

## 5.2 Security Testing

For each policy (cf. §3.4) we determine the operation that may cause a policy violation if used with a specific parameter. We execute these operations several hundred times with a parameter from a known set of violating parameters or a random parameter, similar to *Fuzzing* from software security testing. *Cloud Radar* is required to detect a policy violation in the case of a parameter from the violating set, and otherwise no violation should be detected.

In the case of network isolation, a critical operation is UpdatePortGroup that changes the VLAN identifier of a port group to a given one. If the new VLAN identifier is conflicting with an existing identifier of a different tenant or security zone, the policy is violated. A violating VLAN identifier was chosen with a probability of 1/3. For VM placement, the critical operation is CreateVM that creates a new VM on a given host. The policy is violated if the given host is not part of the same placement zone as the new VM. Finally, storage isolation is violated if a VM is reconfigured (ReconfigVM) with a virtual disk that uses as backend a file already in use by another VM of a different zone.

The security testing uses the real environment described in §4.1, because the simulated one does not support all management operations and its networking configuration is not suited for the network isolation policy. This is not problematic as analysis performance and scalability are not a concern in this security testing, and a real environment yields more realistic behaviour as a simulated environment.

For the network isolation policy, *Cloud Radar* in fact detected all expected violating operations as policy violations, and operations with random operations as non-violations. Overall we issued 254 violating operations and 746 non-violating ones. For the VM placement policy, the system exhibits correct behaviour by detecting 491 violating VM creation operations and reported no violations for 509 non-critical operations. Finally, 505 operations out of 777 VM storage operations have been correctly identified as violations, and for the others the tool correctly reported no violations.

# 6. RELATED WORK

**Configuration Changes in Networks:** Misconfigurations in networks have been a problem in the operation of IT environments for a long time and solutions to mitigate such misconfiguration by the means of change monitoring and analysis have been proposed. Mahajan et al. [10] studied misconfigurations in BGP routing configuration changes by listening to changes and assess these. Kim et al. [9] analysed the evolution of network configurations by mining a repository of network configuration files. With the rise of software-defined networking, real-time monitoring and policy checking have been achieved in these environments [7, 8]. In virtualized infrastructures, misconfiguration may not only happen to the network configuration, but span the entire field of compute, network, and storage resources, which we tackle with *Cloud Radar*.

**Security of Static Virtualized Infrastructures:** The modelling and security analysis of *static* virtualized infrastructures have been subject of existing works [2, 3]. However, these efforts lack the ability to handle dynamic behaviour of such environments. With *Cloud Radar* we are closing this gap by introducing the continuous monitoring of dynamic infrastructures and maintaining an up-to-date model. In fact, our performance evaluation showed significant performance improvements compared to the existing approach.

**Monitoring of Dynamic Virtualized Infrastructures:** Closest to our work is *vQuery* [14] which monitors configuration changes in VMware environments and assess these changes with regard to performance implications. The different goals and motivations of vQuery and *Cloud Radar* (performance versus security) are reflected in the model and configuration translation, where vQuery models many performance metrics and *Cloud Radar* focuses on capturing the topology and its security. Schiffman et al. [13] proposed a monitoring system called *Cloud Verifier* that allows to monitor hosts and virtual machines with regard to integrity requirements, e.g., based on trusted computing mechanisms.

*Cloud Radar* on the other hand focuses on topological properties of the virtualized infrastructure and allows a wide variety of infrastructure security policies to be analysed.

# 7. CONCLUSIONS AND FUTURE WORK

In this paper we presented *Cloud Radar*, a system that detects security failures in virtualized infrastructures in near real-time. The system monitors virtualized infrastructures for changes and based on these changes maintains a graph model of the infrastructure. The model is the input to a model-based security analysis on the infrastructure's topology. The analysis computes and maintains an information flow graph for the dynamic infrastructure, in order to determine isolation properties, and tries find violations of specified security policies. We implemented a prototype of *Cloud Radar* for VMware environments and our performance evaluation shows a significant performance improvement of our event-based approach compared to an existing one that uses static configuration snapshots. The snapshot approach requires $693s$ in a 10,000 VM simulated infrastructure for extracting the configuration and building the models, whereas our event-based one only requires $1.8s$ for each change event after an initialisation of $819s$.

As future work, we aim for further optimisations of the differential information flow analysis by implementing a dynamic SCC computation algorithm. We perceive interesting opportunities along the lines of correlating events with operations and the historical analysis over the evolution of the model. Further, we also plan to extend the model to cope with dynamic access control configurations.

## Acknowledgments

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



# 8. REFERENCES

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

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

[3] 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.

[4] CSA. The Notorious Nine: Cloud Computing Top Threats in 2013. Tech. rep., Cloud Security Alliance (CSA), feb 2013.

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

[6] 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).

[7] Kazemian, P., Chang, M., Zeng, H., Varghese, G., McKeown, N., and Whyte, S. Real Time Network Policy Checking Using Header Space Analysis. In *10th USENIX Symposium on Networked Systems Design and Implementation* (2013), USENIX, pp. 99–111.

[8] Khurshid, A., Zou, X., Zhou, W., Caesar, M., and Godfrey, P. B. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *10th USENIX Symposium on Networked Systems Design and Implementation* (2013), USENIX, pp. 15–27.

[9] Kim, H., Benson, T., Akella, A., and Feamster, N. The Evolution of Network Configuration: A Tale of Two Campuses. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference* (2011), IMC '11, ACM, pp. 499–514.

[10] Mahajan, R., Wetherall, D., and Anderson, T. Understanding BGP Misconfiguration. In *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (2002), SIGCOMM '02, ACM, pp. 3–16.

[11] Ristenpart, T., Tromer, E., Shacham, H., and Savage, S. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *Proceedings of the 16th ACM conference on Computer and communications security* (2009), ACM, pp. 199–212.

[12] Roditty, L., and Zwick, U. A Fully Dynamic Reachability Algorithm for Directed Graphs with an Almost Linear Update Time. In *Proceedings of the Thirty-sixth Annual ACM Symposium on Theory of Computing* (2004), STOC '04, ACM, pp. 184–191.

[13] Schiffman, J., Sun, Y., Vijayakumar, H., and Jaeger, T. Cloud Verifier: Verifiable Auditing Service for IaaS Clouds. In *Proceedings of the IEEE 1st International Workshop on Cloud Security Auditing (CSA 2013)* (June 2013).

[14] Shafer, I., Gylfason, S., and Ganger, G. R. vQuery: a Platform for Connecting Configuration and Performance. *VMware Technical Journal 1*, 2 (Dec. 2012).

[15] Tarjan, R. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing* (1972).

[16] VMware. vSphere 5.5 API Reference, Sep 2013. `http://pubs.vmware.com/vsphere-55/index.jsp#com.vmware.wssdk.apiref.doc/right-pane.html`.

[17] VMware. vSphere Security, ESXi 5.5, vCenter Server 5.5 (EN-001164-04), 2013.