Catching Falling Dominoes: Cloud Management-Level Provenance Analysis with Application to OpenStack

Azadeh Tabiban[1], Yosr Jarraya[2], Mengyuan Zhang[2], Makan Pourzandi[2], Lingyu Wang[1], Mourad Debbabi[1]

[1] CIISE, Concordia University, Montreal, QC, Canada
{a_tabiba, wang, debbabi}@encs.concordia.ca
[2] Ericsson Security Research, Ericsson Canada, Montreal, QC, Canada
{yosr.jarraya, mengyuan.zhang, makan.pourzandi}@ericsson.com

***Abstract*—The dynamicity and complexity of clouds highlight the importance of automated root cause analysis solutions for explaining what might have caused a security incident. Most existing works focus on either locating malfunctioning clouds components, e.g., switches, or tracing changes at lower abstraction levels, e.g., system calls. On the other hand, a management-level solution can provide a big picture about the root cause in a more scalable manner. In this paper, we propose DOMINOCATCHER, a novel provenance-based solution for explaining the root cause of security incidents in terms of management operations in clouds. Specifically, we first define our provenance model to capture the interdependencies between cloud management operations, virtual resources and inputs. Based on this model, we design a framework to intercept cloud management operations and to extract and prune provenance metadata. We implement DOMINOCATCHER on OpenStack platform as an attached middleware and validate its effectiveness using security incidents based on real-world attacks. We also evaluate the performance through experiments on our testbed, and the results demonstrate that DOMINOCATCHER incurs insignificant overhead and is scalable for clouds.**

I. INTRODUCTION

Cloud computing has been widely adopted to provide users the ability to self-provision resources while optimally sharing the underlying physical infrastructure. However, the self-service and multi-tenancy nature of clouds also leads to a higher complexity and greater chances of misconfigurations [5], [17], [35], which may complicate many security issues in clouds. In particular, explaining what may have caused a security incident, i.e., *the root cause analysis*, becomes far more challenging [6]. A manual approach to root cause analysis is typically impractical considering the sheer size of clouds, and automated solutions become essential for understanding, debugging, and preventing security attacks exploiting either vulnerabilities or misconfigurations in clouds.

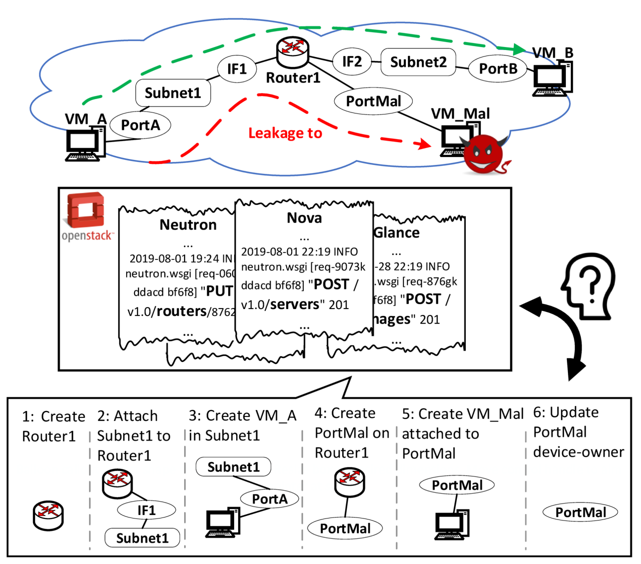
There exist root cause analysis solutions [6], [23], [29] for identifying failed components leading to security alarms in clouds, although they do not explicitly pinpoint the configuration changes causing the failures. Other existing solutions focus on explaining system behaviours through *provenance analysis*, i.e., tracing when and how data objects are created and transformed. However, since most existing provenance solutions work at a low abstraction level, e.g., system calls [11], [25], [26], [34], they become insufficient in the context of clouds, as such solutions would generate a prohibitive amount of provenance metadata while not providing a big picture about the root cause. In the following, we present a motivating example to further highlight the need for provenance analysis in clouds and the limitations of existing solutions.

Motivating Example. Figure 1 depicts the challenge faced by an administrator after the detection of a data leakage from *VM_A* to *VM_Mal* in the cloud virtual infrastructure (shown at the top of the figure), i.e., he/she would have to inspect a large amount of log entries from various services of OpenStack (shown in the middle of the figure) in an attempt to understand the attack scenario (shown at the bottom).

- An attacker from *TenantB* creates a port (*PortMal*) on a router belonging to *TenantA* by exploiting vulnerability `OSSA-2014-008`[1].
- He/She then creates a VM attached to that port while exploiting another vulnerability (`OSSA-2015-018`[2]) to bypass anti-spoofing rules for this VM in order to launch DHCP spoofing attack to impersonate a DNS server.
- He/She now can intercept *TanantA*'s traffic from *VM_A* destined to *VM_B* through *Subnet1*, *Router1* and *Subnet2*.

Pinpointing such attack steps and correlating them based on their interdependencies can be a daunting task if done manually, e.g., at first glance there may not be any apparent link between the *VM_A* creation and the *VM_Mal* attachment to *PortMal*. On the other hand, traditional provenance-based solutions do not directly provide such a big picture, as they typically focus on low-level details (e.g., system calls) of individual components (e.g., an OS). Additionally, interpreting and correlating such low-level results in a cloud would be prohibitive considering its sheer scale.

[1]https://security.openstack.org/ossa/OSSA-2014-008
[2]https://security.openstack.org/ossa/OSSA-2015-018



```
...
2019-08-01 19:24 INI
neutron.wsgi [req-060
ddacd bf6f8] "PUT
v1.0/routers/8762
...
```

```
...
2019-08-01 22:19 INFO
neutron.wsgi [req-9073k
ddacd bf6f8] "POST /
v1.0/servers" 201
...
```

```
...
-28 22:19 INFO
.wsgi [req-876gk
f6f8] "POST /
hages" 201
...
```

Fig. 1: An example of data leakage in clouds (top), logs of various OpenStack services (middle), and the challenge of identifying problematic management operations (bottom).

To address those challenges, we propose in this paper DominoCatcher, a scalable provenance-based solution for forensic analysis in clouds. Our key idea is to lift the provenance analyses to the cloud management-level, which enables tracing cloud infrastructure configuration changes and identifying the ones causing attacks. Specifically, we first design a provenance model to encode the interdependencies between management operations, virtual resources and inputs in clouds. We also propose a middleware-based framework to capture provenance metadata from different cloud services and construct the provenance graph according to our model to support forensic analysis. Finally, we implement and experimentally evaluate a prototype of DominoCatcher on a real OpenStack cloud testbed.

In summary, our main contributions are as follows.

- To the best of our knowledge, this is the first provenance solution focusing on management operations of cloud infrastructures. Compared to existing provenance-based solutions, our provenance model is defined at a higher abstraction level, and therefore, can provide a big picture about cloud configuration changes with increased interpretability that facilitates subsequent analyses.
- In lifting provenance analysis to management-level, we propose several novel mechanisms as follows. First, our middleware-based solution can allow for incremental provenance graph construction, while requiring less instrumentation compared to existing solutions. Moreover, our user-oriented pruning techniques can enable different cloud tenants to customize their analysis of provenance data and to facilitate their different needs in terms of forensic analyses, security assumptions and user preferences.
- Our evaluation results show that DominoCatcher can provide a scalable tool for diagnosing the root cause of security incidents in cloud infrastructures with insignificant performance and storage overhead.

The remainder of this paper is organized as follows: Section II defines our provenance model and Section III describes our methodology. Section IV details our implementation and Section V presents evaluation results. We review related work in Section VI and conclude the paper in Section VII.

II. Cloud Management Provenance Model

We provide a threat model and some background on cloud virtual infrastructures and management operations. We then define our management-level provenance model.

A. Threat Model and Assumptions

Our in-scope threats include both external attackers who exploit existing vulnerabilities in the cloud infrastructure management systems, and insiders, such as cloud users and tenant administrators, who make the state of the cloud infrastructure exploitable either through mistakes or by malicious intentions. We limit our scope to attacks that involve some operations directed through the cloud management interfaces (e.g., command line and dashboard). We assume the cloud infrastructure management system, the provenance building mechanism and the provenance storage are all protected with existing techniques such as remote attestation [12], [30], hash-chain-based provenance storage protection [8] or type enforcement [2].

Out-of-scope threats include attacks that either involve no management operations or can completely bypass the cloud management interfaces, attackers who can temper with (either through attacks or by using insider privileges) the cloud infrastructure management system (e.g., breaching the integrity of the API calls) or the provenance solution itself. Finally, although our provenance results may subsequently lead to the discovery of existing vulnerabilities or misconfigurations, our focus is not on vulnerability analysis, intrusion detection, or configuration verification, and our solution is expected to work in tandem with those solutions.

B. Background

Figure 2 shows an example cloud virtual infrastructure, with cloud tenants provisioning and managing their virtual resources (e.g., VMs and virtual subnets[3]) through API management interfaces (without loss of generality, our running example will focus on virtual network-related security incidents).

Cloud Virtual Infrastructure. As shown in Figure 2, in the cloud virtual infrastructure, routers interconnect different subnets to route intra-tenant traffic (e.g., between *Subnet1* and *Subnet2*), and they also route inter-tenant traffic through external networks. A subnet (e.g., *Subnet1*) is associated with a CIDR (e.g., 10.0.0.0/24) and upon tenants' `Attach-Subnet-to-Router` request, it can be attached to a router through an interface, e.g., *IF1*. Once a tenant requests for creating a VM, e.g., *VM_A*, it is attached to a virtual port, e.g., *PortA*. Ports can be created in subnets and

[3]Different cloud platforms may use different terms for the same concept.

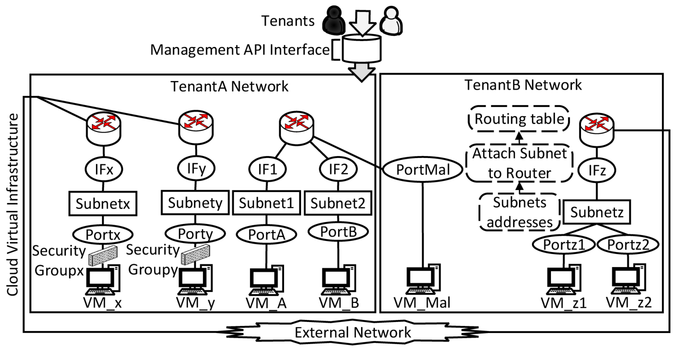


Fig. 2: Example of a cloud virtual infrastructure showing the interdependencies between virtual resources introduced by management operations.

each port is subsequently allocated with an IP address chosen from that subnet's address range. Moreover, ports are attached with one or several security groups, which are the placeholders of access rules specifying the allowed ingress/egress traffic from/to VMs of other groups. Once a tenant requests for attaching a VM to a security group, the iptable of that VM is updated with the VMs' IP addresses from/to which traffic is allowed according to the newly attached security group.

Interdependencies. From the above description about how API calls may affect cloud infrastructures, we can see that there may exist interdependencies between different cloud virtual resources that are introduced by management operations. For instance, the `Attach-Subnet-to-Router` management operation introduces an interdependency between a router and its attached subnets by adding the attached subnets' addresses to the router's routing table, and the `Add-Security-Group` management operation introduces an interdependency between the VMs attached to different security groups. To capture such interdependencies, we define our provenance model in the following.

C. Management-Level Provenance Model

In general, *provenance* usually refers to a technique that captures the information flow between sources and sinks [11]. In the context of cloud virtual infrastructures, we identify as sinks the management operations (e.g., `Create-VM` and `Update-port`) that lead to changing the configuration and state of some virtual resources (e.g., virtual machines and ports), which we identify as sources.

To represent our provenance model, we leverage W3C PROV-DM [4]. According to this specification, the provenance concept is generally visualized using a directed graph, namely *provenance graph*, in which nodes are categorized into three main types: *entities*, *activities*, and *agents*, where entities represent data objects, activities represent transformations on those objects and agents represent software, persons or organizations on whose behalf activities are requested. Relations are defined between nodes to describe their interdependencies, e.g., an entity *WasGeneratedBy* an activity, an activity *Used* an entity, or an activity *WasAssociatedWith* an agent.

To define our provenance model based on PROV-DM, we map the most common concepts in cloud virtual infrastructure management to this specification. Specifically, a summary of our provenance model is shown in Table I. Subtypes are added to refine the classification of PROV-DM based on our needs. To illustrate our model, Figure 3 shows an excerpt of the provenance graph describing the management operations involved in our motivating example, as detailed in the following.

Entities. As explained in Table I, entity vertices (shown as ovals in Figure 3) represent states of cloud virtual infrastructure resources, their configuration or inputs (e.g., a virtual router, security group, VM, etc.). For instance, a state of a router can be associated with the addresses of its connected networks, while a VM state can be either *running* or *down*. We use node versioning, which is the most common cycle removal technique in the provenance literature [21], [25], [26], in order to reduce the subsequent analyses overhead. Specifically, a new node is created at each change occurrence of its corresponding resource, representing a new version of the resource[4]. For example, as it is shown in Figure 3, a new node representing an updated state of *Router1* (i.e., the node $\langle$*Router1*, *Version1*$\rangle$) is created when it is attached to *Subnet1*, which essentially represents the updated routing table. Each entity node is assigned with a label describing its subtype (e.g., VM, Port, etc.). Moreover, entity nodes consist of a set of attributes for storing a unique ID assigned by the cloud to their resources, nodes' creation time, etc. Other attributes, such as attached networks for virtual routers or running/stopped for VMs, may also be assigned when needed.

Activities. Activity vertices (shown as rectangles in Figure 3) represent management API calls made to either change the state of resources (e.g., `Start-VM`) or to mange their lifecycle (e.g., `Create-VM`). The management API calls can be made either directly by tenants or as the result of another operation request. For example, in OpenStack, once a tenant requests for creating a VM in a network, his/her request is received by the compute service, which subsequently makes another request to the networking service for binding a port in the specified network to the created VM. In such a case, we consider those two API calls as separate activities. We assign each activity node a label describing its corresponding operation type, e.g., `Create-VM`. Moreover, each activity node has several attributes, including a unique request ID assigned by the cloud management system to its corresponding API call, the time that the request has been issued, etc.

Agents. Agent vertices (shown as diamonds in Figure 3) correspond to the identity of the tenant admins or users interacting with management API interfaces to provision or manage their resources. Agents are identified using the unique ID of tenants or users.

[4]Although node versioning naturally causes an increase in the size of the provenance graph, we will show that the size of our provenance graph is sufficiently scalable in Section V.

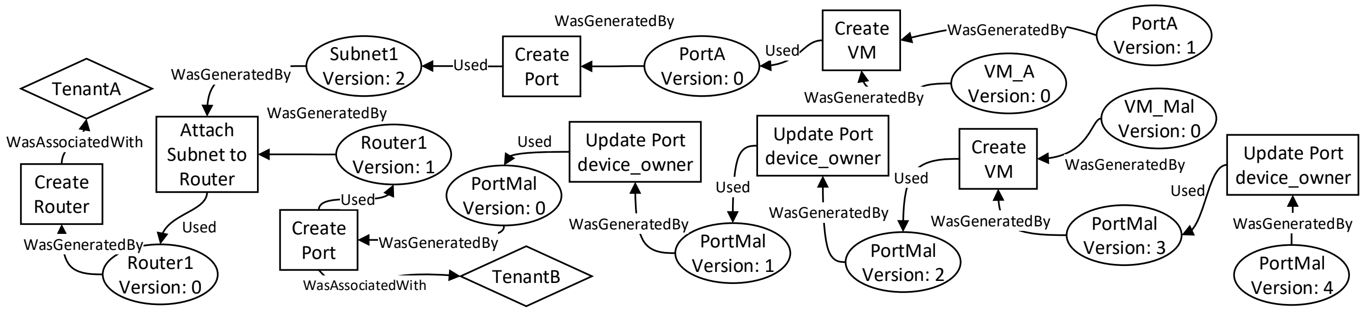


Fig. 3: The provenance graph describing the main management operations and virtual resources related to the attack in the motivating example (some edges and nodes are omitted for the sake of readability).

TABLE I: Mapping of the common concepts in cloud virtual infrastructures to the PROV-DM Model.

Cloud Concept	Description	PROV Model	Subtype
Cloud Tenant	A group of users owning an isolated set of virtual resources.	Agent	Tenant Admin, other tenants
Cloud User	Customer of the cloud infrastructure belonging to a tenant with specific privileges to provision cloud resources.	Agent	Admin user of a tenant, other users.
Operation (lifecycle-related)	Management API calls for deploying, deleting, or updating cloud virtual resources.	Activity	Create-VM, Update-Port-Device-Owner, Update-VLAN-ID, etc
Operation (state-related)	Management API calls for performing actions on virtual resources.	Activity	Start VM, Resize VM, Change VM Password, etc.
Resource	The states of a virtual infrastructure resource. For example, a VM is running/stopped/paused.	Entity	VMs, virtual ports, virtual Subnets, etc.
Resource Configuration	The state of a virtual infrastructure configuration, e.g., configuration state of the virtual hardware for VMs, network access rules, etc.	Entity	Security groups, Flavors, etc.
Input for changing configurations	An input data causing a change to the configuration state.	Entity	Security group rules, etc.

III. The Methodology

We first provide an overview of our methodology, and then detail the provenance construction and forensic analysis stages.

A. Overview

An overview of the DominoCatcher framework is shown in Figure 4. DominoCatcher works in two main stages, i.e., provenance construction (represented as solid line arrows) and offline forensics analysis (dashed line arrows). First, DominoCatcher intercepts tenants' management API calls at runtime to incrementally construct the provenance graph. During provenance construction, DominoCatcher intercepts and processes each API call to construct the corresponding subgraph and then appends it to the rest of the provenance graph in the database. Once a threat is detected, the investigator can trigger the DominoCatcher offline forensic analysis capabilities to perform an algorithmic pruning on the provenance graph in order to narrow down the cause of the threat. We detail those two stages in the following.

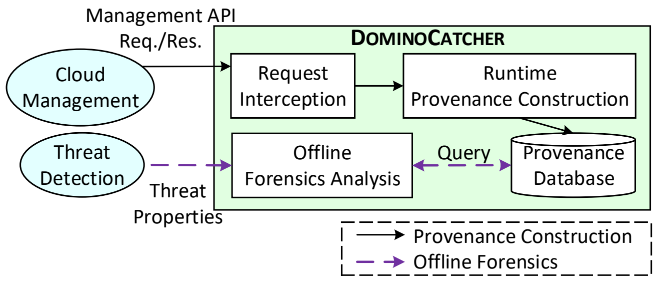


Fig. 4: DominoCatcher highlevel overview.

B. Provenance Construction

Provenance construction consists of two main steps: data collection and graph generation.

Data Collection. Existing provenance-based solutions mostly perform data collection through log processing, event interception mechanisms via Linux Security Modules (LSM) [20], or code instrumentation in other platforms. Specifically, these interception mechanisms trace the interdependencies between data objects [7], [25], [26] or application functions and variables [33], [34]. However, code instrumentation would be too expensive and impractical in cloud infrastructures considering their sheer scale and complexity. On the other hand, standard cloud infrastructure logs may lack sufficient details for identifying the type of management operations and corresponding resources required for provenance metadata (e.g., different types of operations may appear to be identical in the logs [18]).

To address those limitations, we design our event interception mechanism as a middleware [14], [31] to intercept the API calls. This approach allows us to trace more detailed information about configuration changes (than what is included in the logs), while avoiding the expensive instrumentation and decoupling the provenance system from the infrastructure for more flexible deployment. The intercepted API calls are processed by DominoCatcher according to rules built based on the cloud API design. These rules are used to parse the API calls, identify the type of management operations, determine the affected virtual resources and the user identity behind the API request. More details on parsing the intercepted requests and retrieving their parameters will be provided in Section IV.

Graph Generation. After data collection, DominoCatcher converts the extracted information into provenance metadata as nodes and edges, and appends them to the provenance graph stored in the database. Specifically, it first creates a new node for each affected virtual resource and a node for the requested operation. Next, it creates relations such as a *WasGeneratedBy* edge from each resource node to the operation node, or a *Used* edge from the operation node to each of the existing nodes in the provenance graph that represents the latest version of an affected resource, which is identified through its unique ID and the previous versions' timestamps.

For example, in Figure 3, DOMINOCATCHER creates a *WasGeneratedBy* edge from the node ⟨*Router1*, *Version1*⟩ (representing its state after *Subnet1*'s attachment), to *Attach-Subnet-to-Router* and a *Used* edge from *Attach-Subnet-to-Router* to the node ⟨*Router1*, *Version0*⟩, representing the previous state of *Router1*. Furthermore, DOMINOCATCHER creates a *WasAssociatedWith* edge from the operation node (e.g., *Create-Router*) to the node representing the cloud user/tenant requesting that operation (e.g., *TenantA*).

C. Forensic Analysis

To explain what might have led to attacks in the cloud virtual infrastructure, analysts could perform forensic analyses on the provenance graph constructed by DOMINOCATCHER prior to the time of threat detection. To this end, we can leverage existing threat detection mechanisms for monitoring the infrastructure and the deployed VMs. For instance, we can rely on three main types of detection methods: VM-level monitoring [3] (e.g., intrusion detection tools), cloud virtual infrastructure policy compliance [24] and cross-layer consistency verification tools [15].

In performing forensic analyses on the provenance graph constructed by DOMINOCATCHER, analysts may face two challenges. First, the provenance graph may be too large for human interpretation as it might include many benign or irrelevant operations. Second, the multi-tenancy nature of clouds means the analysts from different tenants may have vastly different needs and preferences in terms of their objectives of forensic analyses and security assumptions. To address those issues, we propose two user-oriented pruning mechanisms to automatically identify and remove benign or irrelevant information from the provenance graph. The analysts could narrow down the scope of their forensic analyses by triggering the pruning process and selecting the proper pruning mechanisms based on their needs.

Specifically, we propose two user-oriented pruning schemes in the context of cloud virtual infrastructures, namely *disjoint subgraph* pruning and *context-based* pruning (we can additionally apply other pruning approaches, e.g., [7], [9], [10]).

Disjoint Subgraph Pruning. This pruning mechanism basically finds and removes all nodes in the provenance graph that are disconnected from the subgraph including the node corresponding to the target resource (i.e., the resource on which a threat is detected). Specifically, DOMINOCATCHER starts from the last version of the breached resource node, and follows all paths of the type *Resource1-(Used/WasGeneratedBy)-ManagementOperation-(WasGeneratedBy/Used)-Resource2*. An example provenance graph with this pruning mechanism applied is depicted in Figure 5. In the figure, starting from the starred *VMA* node, we can find the *Add-Security-Group* node, which *used* the

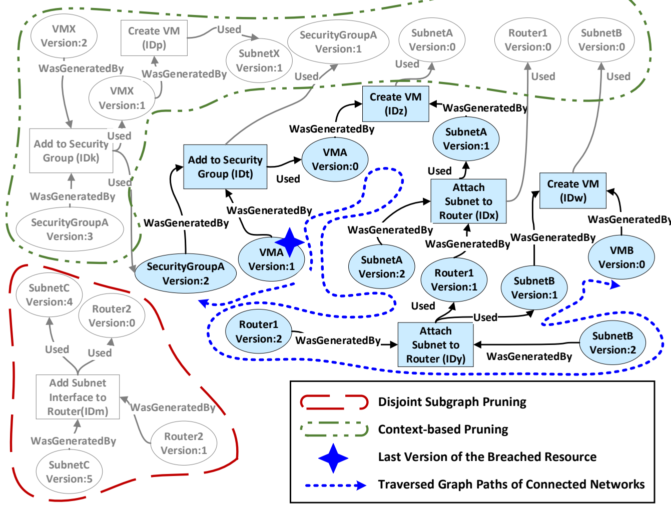


Fig. 5: Example of pruning through finding the provenance graph nodes disjointed from the target VM (*VMA*) and unrelated nodes according to the security incident context.

previous version of *VMA* as well as the new version of *SecurityGroupA*. We further follow the operations affected *VMA* earlier in the provenance graph, which leads us to the *SubnetA* node through the *WasGeneratedBy* edge from the *Create-VM* node. We also find the *Attach-Subnet-to-Router* node that *used SubnetA* and *generated* a new version of *Router1*. Following this process, we can find all the operations and resources that have affected the state of the target resource, *VMA*, as well as the operations and resources which are dependent on the changes previously made to *VMA*. On the other hand, the absence of any path between a group of nodes and the target resource implies the lack of interdependencies and thus the former may be pruned. For example, in Figure 5, having detected an attack on *VMA*, an analyst may trigger this pruning mechanism to prune the disjoint subgraph, shown inside the dashed contour in red, e.g., *SubnetC*, *Router2* and the operations affecting them.

Context-Based Pruning. This pruning mechanism removes nodes that are not contextually dependent on the target resource according to the analyst's provided criteria. DOMINOCATCHER traverses paths in the provenance graph while checking the specified constraints to identify a subgraph of resources and operations interdependent with the target virtual resource. For example, a security incident reported by a network-based IDS can lead the investigator to prune entity and activity nodes that are related to virtual resources not directly connected to the same virtual network as the victim resources. Figure 5 shows an example of context-based pruning in the context of *VMA* data leakage incident. In this example, DOMINOCATCHER automatically identifies resources connected to *VMA*'s network (*SubnetA*) through the provenance graph based on the predefined operation types that potentially create or update network connectivity between virtual resources (e.g., `Create-VM` and

`Attach-Subnet-to-Router`). To this end, it starts from the last version of *VMA* entity node in the provenance graph and traverses paths until it either reaches activity nodes of the operation types not included in the predefined set or not acting on resources that are reachable from *VMA* nodes through any traversed path. For instance, DOMINOCATCHER keeps *VMB* and its attached subnet, *SubnetB*, in the provenance graph as they are reachable from *VMA* through *Attach-Subnet-to-Router* and *Create-VM* nodes on a traversed path. On the other hand, although *VMX*, connected to *SubnetX* through *Create-VM* node (in the green contour), is attached to the same security group as *VMA*'s via *Add-Security-Group* node, it is pruned in this step, as there is no *Attach-Subnet-to-Router* node on any path between the network of *VMX* and *VMA*.

The pruning mechanisms allow the analyst to perform a more focused forensic analysis towards the identification of potential root causes. The following demonstrates how an analyst may pinpoint the management operations that lead to data leakage from *VMA* in our motivating example.

- The analyst may first trigger the disjoint graph pruning mechanism to retrieve the subgraph with nodes reachable from the *VMA* nodes through some paths.
- Next, through the context-based pruning, he/she can retrieve all nodes corresponding to the resources that became routable from *VMA*'s network before and after *VMA*'s creation.
- Based on the pruned result (Figure 3), the analyst may realize that *VMA* was created in a network that was attached to a router, *Router1*, belonging to *TenantA*.
- Among the retrieved nodes, he/she can query to highlight the nodes corresponding to the operations requested by users of a tenant different from *TenantA*, and their affected resources. He/She can see that a user of *TenantB* created a port on the router attached to *VMA*'s network, and later, that user updated the port's device_owner field immediately after he/she created and attached a VM, *VMB*, to that port.
- Seeing the creation of a port by a user on a different tenant's router, the analyst realizes about the existence of an authorization failure. Furthermore, since the update of the port's device_owner field was requested immediately after its attachment to a created VM, it is likely that OpenStack Neutron service treated it as a network-owned port, and thus, assigned no anti-spoofing rules to the port. Those lead the analyst to realize that there may exist vulnerabilities in Neutron which allow users to bypass proper authorization check and anti-spoofing rules to access other tenants' networks.

IV. IMPLEMENTATION

We implement DOMINOCATCHER based on an OpenStack cloud testbed. We choose OpenStack due to its popularity, e.g., as a platform supporting Network Function Virtualization (NFV) for telecommunication service providers [27]. However, we note that only the data collection mechanism of our approach is platform-specific, and our modular and less invasive design makes our approach readily adaptable to other platforms.

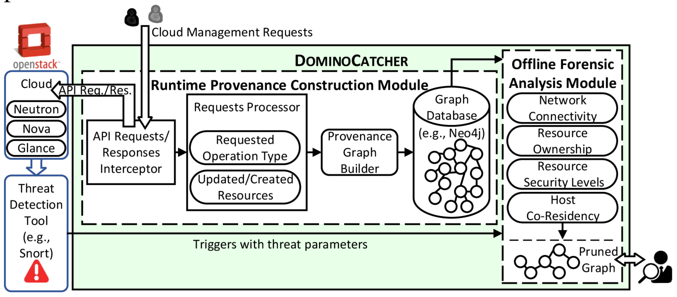

Cloud Management Requests
openstack
Cloud
API Req./Res.
Neutron
Nova
Glance
Threat Detection Tool (e.g., Snort)
DOMINOCATCHER
Runtime Provenance Construction Module
API Requests/ Responses Interceptor
Requests Processor
Requested Operation Type
Updated/Created Resources
Provenance Graph Builder
Graph Database (e.g., Neo4j)
Offline Forensic Analysis Module
Network Connectivity
Resource Ownership
Resource Security Levels
Host Co-Residency
Pruned Graph
Triggers with threat parameters

Fig. 6: DOMINOCATCHER architecture.

Figure 6 shows the architecture of DOMINOCATCHER. In the following, we detail the implementation and integration of our approach in OpenStack.

Integration into OpenStack. In the following, we explain the integration of DOMINOCATCHER into OpenStack and the preprocessing required for the data collection.

1) DOMINOCATCHER *as an OpenStack WSGI middleware.* To collect provenance metadata from the REST API calls made to endpoint services (e.g., Nova and Neutron), we implement our framework as a Python *WSGI* middleware similar to existing works [14], [31], and install it in the filter chain to those services. Figure 7a depicts an excerpt of Neutron API configuration and its filter chain into which DOMINOCATCHER is inserted. This configuration is stored in the `api-paste.ini` file for each service.

```
[composite:neutronapi_v2_0]
use = call:neutron.auth:pipeline_factory
noauth = cors http_proxy_to_wsgi request_id catch_errors extensions
DominoCatcher neutronapiapp_v2_0
keystone = cors http_proxy_to_wsgi request_id catch_errors authtoken
keystonecontext DominoCatcher extensions neutronapiapp_v2_0
```

(a)

REQUEST_METHOD: "PUT"
openstack.request_id: "234dt"
HTTP_X_PROJECT_ID: "fb5s"
HTTP_X_USER_ID: "ax1h"
PATH_INFO: "/v2.0/ports/f91398"
wsgi.input: "{"port": {"device_owner": "network:--"}}"

(b)

Fig. 7: (a) OpenStack Neutron API configuration integrating DOMINOCATCHER as a middleware. (b) Example of `Update-Port` API call parameters.

2) *Preprocessing for Information Extraction.* Our approach enables users to focus provenance analysis on a set of operations and resources as well as OpenStack services which are specified to be related to the analyses. For instance, the users can focus on the management operations that update network-related cloud configurations. This is achieved through specifying parsing and operation typing rules of the selected management operations to extract the contextual information and to identify the type of the requested operations at runtime. Figure 7b shows selected fields of an example `Update-Port` API call.

In this example, the `PATH-INFO` is parsed to extract the updated port. Also, the extracted fields of `wsgi.input` (the request body content), `METHOD` and `PATH-INFO` of the API call are matched against the typing rules to determine that this request is issued to update port *'f91398'* while changing its device_owner field.

At the end of the preprocessing stage, DOMINOCATCHER is ready to intercept management API calls, extract the affected virtual resources and identify the requested operations' type.

Runtime Provenance Construction. At runtime, *API Requests/Responses Interceptor* intercepts the parameters of management API calls and passes them to *Requests Processor*, which identifies the affected resources and the type of the requested operations. If it identifies a request triggering the creation of resources (e.g., `Create-VM`), it processes the API response sent back from the endpoint services as well to retrieve the created resource ID. Next, *Provenance Builder* updates the provenance database implemented in Neo4j[5] with the extracted information. We use *py2neo*[6] library in DOMINOCATCHER middleware as an interface between the middleware python script and Cypher language[7] to interact with the database.

Offline Forensic Analysis Module. To facilitate the analyses after the detection of a threat, the analyst selects and initializes selected pruning mechanisms (detailed in Section III) through DOMINOCATCHER command line interface. For instance, the analyst can initiate the pruning script with the parameters reported about the alert (e.g., time of the detection, the target VM ID, etc.) and a *beginning time* to further limit the analyses on the constructed graph.

V. EVALUATION

In this section, we evaluate DOMINOCATCHER based on three criteria: (1) Effectiveness in reconstructing the operations sequences that led to the attack; (2) Runtime performance overhead; (3) Storage overhead. We conduct our experiments based on OpenStack *Rocky*[8]. Our cloud testbed includes one controller node and up to 80 compute nodes, each with 8 CPUs and 12 GB RAM running Ubuntu 16.04 server.

A. Effectiveness

To evaluate the effectiveness of our approach, we reproduce in our testbed 8 attack scenarios that involve cloud virtual infrastructure misconfigurations. Most of these attacks are discussed in existing works [5], [15], [17], [31], [35]. Table II summarizes these attack scenarios and the most relevant operation types according to the results obtained after the DOMINOCATCHER pruning process. In the following, we detail how a cloud admin can benefit from DOMINOCATCHER for the case of port scanning threat (other cases are omitted due to page limitations).

[5]https://neo4j.com
[6]https://py2neo.org/v4
[7]https://neo4j.com/developer/cypher-query-language
[8]https://docs.openstack.org/rocky

Example. Consider a scenario where a user (*UserA*) receives a port scanning alert on one of his/her running VMs, *VMA*, which he/she previously connected to *VMB*'s network to use the network service on *VMB* belonging to a trusted user, *UserB*. Figure 8 shows the pruned provenance graph generated by DOMINOCATCHER. For simplicity, we do not show the attachment of *VMB* to *SecurityGroupB* and the rule allowing traffic from this group to *SecurityGroupA*. The provenance graph shows that *VMA* was created in *SubnetA*, and added to *SecurityGroupA* before getting started. Moreover, *SubnetA* was attached to *Router1* through `Attach-Subnet-to-Router` operation, and *Router1* was connected to *SubnetB*, and in *SubnetB*, *UserMal* created *VM_Mal*, and added it to *SecurityGroupX*. We can also see the creation of *SecuirtyGroupRuleX* which allowed ICMP traffic from the VMs of *SecurityGroupX* to the VMs of *SecurityGroupA*, but this rule was deleted after *VMA* got started and before *VMA*'s subnet (*SubnetA*) was connected to *SubnetB* (through their attachment to *Router1*). The fact that the attacker succeeded to send traffic of this type in spite of the traced operations in the provenance graph, makes the admin suspect that the problem was related to the application of the security group rules. Therefore, the admin examines the infrastructure and security groups, and discovers a vulnerability in the networking service (`CVE-2015-7713`[9]) which does not allow the changes of security groups to be reflected immediately on running VMs.

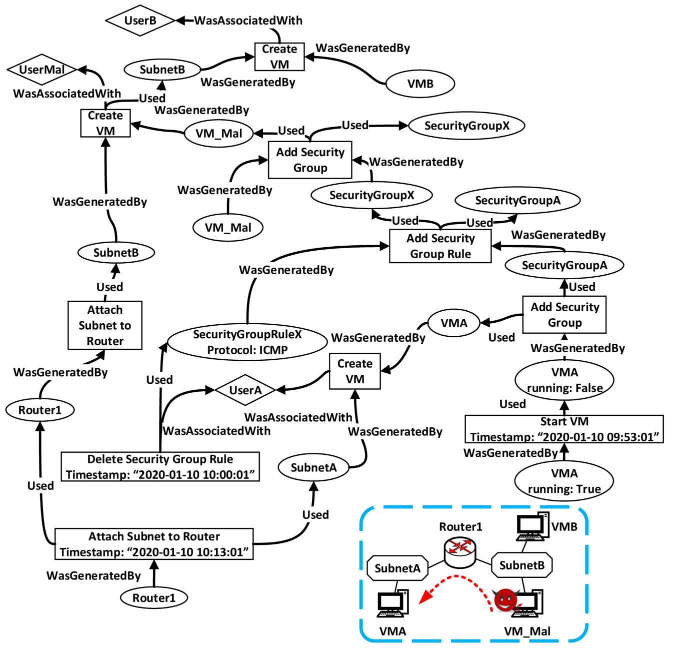


Fig. 8: Diagnosing the root cause of the port scanning threat alert. We remove unnecessary information (e.g., version numbers) for readability purposes.

B. Performance

To evaluate the performance overhead of our approach, we measure the latency in handling management requests (i.e.,

[9]https://security.openstack.org/ossa/OSSA-2015-021

TABLE II: Attack scenarios used to evaluate DOMINOCATCHER effectiveness.

Root Causes	Detected Threat	Most Relevant Management Operation Types	Vulnerability
Malformed security group rule addition	Cross-layer Inconsistency	Create-Security-Group, Create-Security-Group-Rule, Create-VM	CVE-2019-9735
Overlapping security group rule addition	Cross-layer Inconsistency	Create-Security-Group, Create-Security-Group-Rule, Create-VM	CVE-2019-10876
Update of security group is not applied [17]	Port Scanning	Add-Security-Group, Start-VM, Delete-Security-Group-Rule	CVE-2015-7713
Race condition to bypass anti-spoofing rules [31]	Data Leakage	Create-Port, Create-VM, Update-Port	CVE-2015-5240
Entering a different tenant network by router cross-plugged [35]	Data Leakage	Create-Router, Create-Port, Create-VM	CVE-2014-0056
Wrong VLAN ID [5]	Data Leakage	Create-Network, Update-Network	Not specified
Failing to delete VMs in resize state	Disk Consumption	Create-VM, Resize-VM, Delete-VM	CVE-2016-7498
Excessive VM creation on the same host [16]	Disk Consumption	Create-VM	Not specified

the elapsed time between sending a request from management interfaces and the completion of its execution). The additional latency imposed by DOMINOCATCHER consists of the time required for data collection as well as generating the provenance graph incrementally. We note that the time required for the communication[10] between DOMINOCATCHER and Neo4j server for provenance construction is included in our measurements. Our results are measured in more than 50 trials and in three different cloud sizes: 600, 1800 and 3000 VMs with respectively 43069, 64689 and 107936 graph nodes for each cloud size. Figure 9 shows a comparison between the latency of management operations execution (with and without DOMINOCATCHER performing runtime provenance construction) in different cloud sizes. As it is depicted, the provenance construction delay increases with the size of the cloud, which can be justified by their corresponding greater graph size and the elapsed time required for finding the parents of the newly created entity nodes. The total overhead in the cloud with 600, 1800 and 3000 VMs remains around %2.1, %2.5 and %4.1 respectively in more than half of the cases. Those results confirm that our solution is reasonably scalable for clouds.

C. Storage Overhead

We measure the storage cost of DOMINOCATCHER, which is important for the providers to allocate the required storage resources for supporting DOMINOCATCHER. Our results are depicted in Figure 9d. As it is shown, for the provenance graph constructed with 120,000 operations, only 80-megabyte storage is needed. This number of operations is much higher than the number of configuration API calls issued in one day in a real enterprise cloud reported in [35], which indicates the storage cost of DOMINOCATHCER is acceptable.

VI. RELATED WORK

Provenance-based security solutions have been extensively explored in [7], [11], [25], [26]. Many of these approaches are based on tracing system transformations through low-level system calls. For example, King et al. [11] leverage data provenance to explain security incidents by tracing back related events and system components in Unix-like operating systems. To improve the provenance capture mechanism, the authors in [7], [25], [26] build provenance graph based on the information captured by Linux Security Module hooks. Hi-Fi [28] utilizes provenance on kernel-level to monitor malicious behavior within a compromised system, while LPM [2] uses provenance DAGs to ensure authenticated communications. To increase the efficiency of online analyses, CamQuery [26] traces both userspace and in-kernel executions. HOLMES [19] provides a summarized explanation of the attacker's actions based on low-level system calls through removing provenance graph nodes and edges unrelated to attack campaigns. Although most of these solutions can be extended to clouds, in contrast to our work, they cannot directly provide a big picture about the root cause of security incidents, and they also lack the interpretability and scalability of our approach.

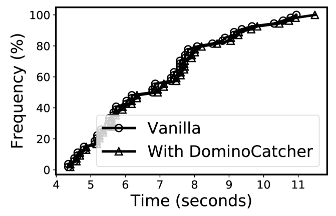
Other recent efforts [32]–[34], [36] adapt provenance analysis to different domains. ProvThings [34] proposes a platform-centric provenance-based approach for auditing the Internet of Things (IoT) applications cross different devices. In SDN environments, FORENGUARD [33] provides flow-level forensics and ProvSDN [32] monitors the access to sensitive data for unprivileged applications through privileged ones. Wu et al. [36] define *negative provenance* to explain the absence of events in distributed systems. Unlike our work, none of these solutions specifically focus on cloud infrastructure management systems.

There exist only limited efforts on applying provenance analysis to cloud virtual infrastructures. Lu et al. [13] propose a forensics schema to investigate the data access and Bates et al. [1] propose to use provenance-based access control mechanism to ensure cloud storage security. The authors in [22], propose a tenant-aware provenance-based solution to enhance OpenStack access control mechanism. In contrast, our approach leverages the provenance concept to trace information flow between virtual resources through management API calls, which allows it to be used for monitoring a wider range of changes in cloud infrastructure systems.

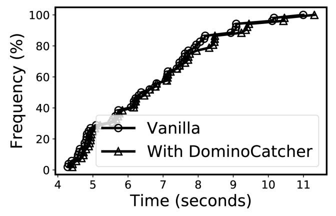
VII. CONCLUSION

In this paper, we presented DOMINOCATCHER, the first management-level provenance solution for clouds. DOMINOCATCHER leveraged data provenance concept to find the management operations leading to attacks in cloud virtual infrastructures and provided efficient pruning mechanisms for users to pinpoint the root causes. We integrated DOMINOCATCHER to OpenStack and demonstrated the efficacy of our approach based on real attack scenarios. Moreover, our experiments on performance and storage cost showed the applicability of our approach with insignificant overhead. As future work, we plan to combine our framework with low-level

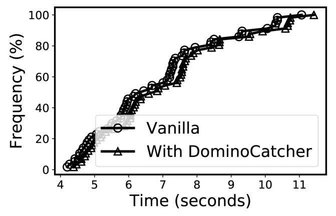
[10]https://neo4j.com/docs/driver-manual/current/client-applications



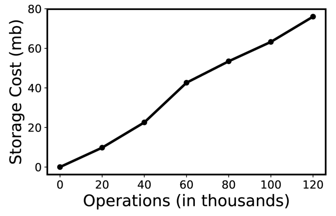
(a)



(b)



(c)



(d)

Fig. 9: Average latency in clouds with 600 (a), 1800 (b) and 3000 VMs (c). Provenance storage growth (d).

provenance-based techniques to cover potential gaps between operational models and actual implementations in clouds (e.g., to trace changes bypassing API interfaces). Furthermore, we will use machine learning techniques to further facilitate the identification of operations responsible for the detected attacks.

Acknowledgment

We thank the anonymous reviewers for their valuable comments and suggestions. This work was supported partially by the Natural Sciences and Engineering Research Council of Canada and Ericsson Canada under the Industrial Research Chair (IRC) in SDN/NFV Security.

References

[1] Bates, A., Mood, B., Valafar, M., and Butler, K. R. B. Towards secure provenance-based access control in cloud environments. In *CODASPY* (2013), pp. 277–284.

[2] Bates, A., Tian, D. J., Butler, K. R., and Moyer, T. Trustworthy whole-system provenance for the linux kernel. In *USENIX Security* (2015), pp. 319–334.

[3] Bauman, E., Ayoade, G., and Lin, Z. A survey on hypervisor-based monitoring: approaches, applications, and evolutions. *ACM Computing Surveys (CSUR) 48*, 1 (2015), 10.

[4] Belhajjame, K., B'Far, R., Cheney, J., Coppens, S., Cresswell, S., Gil, Y., Groth, P., Klyne, G., Lebo, T., McCusker, J., et al. Prov-dm: The prov data model. *W3C Recommendation. http://www. w3. org/TR/prov-dm* (2013).

[5] Bleikertz, S., Vogel, C., Gross, T., and Mödersheim, S. Proactive security analysis of changes in virtualized infrastructures. In *ACSAC* (2015), ACM, pp. 51–60.

[6] Cisco. Cisco avos. Available at: https://github.com/CiscoSystems/avos, last accessed on: January 14, 2020.

[7] Han, X., Pasquier, T., Ranjan, T., Goldstein, M., and Seltzer, M. Frappuccino: Fault-detection through runtime analysis of provenance. In *HotCloud* (2017).

[8] Hasan, R., Sion, R., and Winslett, M. The case of the fake picasso: Preventing history forgery with secure provenance. In *FAST* (2009).

[9] Hassan, W. U., Aguse, L., Aguse, N., Bates, A., and Moyer, T. Towards scalable cluster auditing through grammatical inference over provenance graphs. In *NDSS* (2018).

[10] Hassan, W. U., Guo, S., Li, D., Chen, Z., Jee, K., Li, Z., and Bates, A. Nodoze: Combatting threat alert fatigue with automated provenance triage. In *NDSS* (2019).

[11] King, S. T., and Chen, P. M. Backtracking intrusions. In *SOSP* (2003), pp. 223–236.

[12] Li, M., Zang, W., Bai, K., Yu, M., and Liu, P. Mycloud: supporting user-configured privacy protection in cloud computing. In *ACSAC* (2013), ACM, pp. 59–68.

[13] Lu, R., Lin, X., Liang, X., and Shen, X. S. Secure provenance: the essential of bread and butter of data forensics in cloud computing. In *ASIA CCS* (2010), pp. 282–292.

[14] Luo, Y., Luo, W., Puyang, T., Shen, Q., Ruan, A., and Wu, Z. Openstack security modules: A least-invasive access control framework for the cloud. In *IEEE CLOUD* (2016), pp. 51–58.

[15] Madi, T., Jarraya, Y., Alimohammadifar, A., Majumdar, S., Wang, Y., Pourzandi, M., Wang, L., and Debbabi, M. Isotop: Auditing virtual networks isolation across cloud layers in openstack. *ACM Transactions on Privacy and Security (TOPS) 22*, 1 (2018), 1.

[16] Madi, T., Zhang, M., Jarraya, Y., Alimohammadifar, A., Pourzandi, M., Wang, L., and Debbabi, M. Quantic: Distance metrics for evaluating multi-tenancy threats in public cloud. In *CloudCom* (2018), IEEE, pp. 163–170.

[17] Majumdar, S., Jarraya, Y., Oqaily, M., Alimohammadifar, A., Pourzandi, M., Wang, L., and Debbabi, M. Leaps: Learning-based proactive security auditing for clouds. In *ESORICS* (2017), Springer, pp. 265–285.

[18] Majumdar, S., Tabiban, A., Jarraya, Y., Oqaily, M., Alimohammadifar, A., Pourzandi, M., Wang, L., and Debbabi, M. Learning probabilistic dependencies among events for proactive security auditing in clouds. *Journal of Computer Security*, Preprint (2019), 1–38.

[19] Milajerdi, S. M., Gjomemo, R., Eshete, B., Sekar, R., and Venkatakrishnan, V. N. HOLMES: real-time APT detection through correlation of suspicious information flows. In *IEEE S&P* (2019), pp. 1137–1152.

[20] Morris, J., Smalley, S., and Kroah-Hartman, G. Linux security modules: General security support for the linux kernel. In *USENIX Security* (2002), ACM Berkeley, CA, pp. 17–31.

[21] Muniswamy-Reddy, K.-K., Holland, D. A., Braun, U., and Seltzer, M. I. Provenance-aware storage systems. In *USENIX Annual Technical Conference, General Track* (2006), pp. 43–56.

[22] Nguyen, D., Park, J., and Sandhu, R. Adopting provenance-based access control in openstack cloud iaas. In *International Conference on Network and System Security* (2014), Springer, pp. 15–27.

[23] OpenStack. Vitrage (rca (root cause analysis) service). Available at: https://governance.openstack.org/tc/reference/projects/vitrage.html, last accessed on: January 14, 2020.

[24] OpenStack. OpenStack Congress, 2015. Available at: https://wiki.openstack.org/wiki/Congress, last accessed on: February 1, 2019.

[25] Pasquier, T., Han, X., Goldstein, M., Moyer, T., Eyers, D., Seltzer, M., and Bacon, J. Practical whole-system provenance capture. In *SoCC* (2017), ACM, pp. 405–418.

[26] Pasquier, T., Han, X., Moyer, T., Bates, A., Hermant, O., Eyers, D., Bacon, J., and Seltzer, M. Runtime analysis of whole-system provenance. In *CCS* (2018), ACM, pp. 1601–1616.

[27] Pattaranantakul, M., He, R., Song, Q., Zhang, Z., and Meddahi, A. Nfv security survey: From use case driven threat analysis to state-of-the-art countermeasures. *IEEE Communications Surveys & Tutorials 20*, 4 (2018), 3330–3368.

[28] Pohly, D. J., McLaughlin, S. E., McDaniel, P. D., and Butler, K. R. B. Hi-fi: collecting high-fidelity whole-system provenance. In *ACSAC* (2012), pp. 259–268.

[29] Samir, A., and Pahl, C. A controller architecture for anomaly detection, root cause analysis and self-adaptation for cluster architectures. In *Intl Conf on Adaptive and Self-Adaptive Systems and Applications* (2019).

[30] Schear, N., Cable II, P. T., Moyer, T. M., Richard, B., and Rudd, R. Bootstrapping and maintaining trust in the cloud. In *ACSAC* (2016), ACM, pp. 65–77.

[31] Tabiban, A., Majumdar, S., Wang, L., and Debbabi, M. Permon: An openstack middleware for runtime security policy enforcement in clouds. In *CNS* (2018), IEEE, pp. 1–7.

[32] Ujcich, B. E., Jero, S., Edmundson, A., Wang, Q., Skowyra, R., Landry, J., Bates, A., Sanders, W. H., Nita-Rotaru, C., and Okhravi, H. Cross-app poisoning in software-defined networking. In *CCS* (2018), ACM, pp. 648–663.

[33] Wang, H., Yang, G., Chinprutthiwong, P., Xu, L., Zhang, Y., and Gu, G. Towards fine-grained network security forensics and diagnosis in the sdn era. In *CCS* (2018), ACM, pp. 3–16.

[34] Wang, Q., Hassan, W. U., Bates, A., and Gunter, C. Fear and logging in the internet of things. In *NDSS* (2018).

[35] Wang, Y., Madi, T., Majumdar, S., Jarraya, Y., Alimohammadifar, A., Pourzandi, M., Wang, L., and Debbabi, M. Tenantguard: Scalable runtime verification of cloud-wide vm-level network isolation. In *NDSS* (2017).

[36] Wu, Y., Zhao, M., Haeberlen, A., Zhou, W., and Loo, B. T. Diagnosing missing events in distributed systems with negative provenance. In *ACM SIGCOMM* (2014), pp. 383–394.