Provenance Analysis in Virtualized Environments

Azadeh Tabiban

A Thesis
in
The Concordia Institute
for
Information Systems Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of
Doctor of Philosophy (Information and Systems Engineering) at
Concordia University
Montréal, Québec, Canada

September 2022

© Azadeh Tabiban, 2022

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: **Azadeh Tabiban**
Entitled: **Provenance Analysis in Virtualized Environments**

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Information and Systems Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

Dr. Sivakumar Narayanswamy	Chair
Dr. Frédéric Cuppens	External Examiner
Dr. Ferhat Khendek	External to Program
Dr. Mohsen Ghafouri	Internal Examiner
Dr. Suryadipta Majumdar	Internal Examiner
Dr. Lingyu Wang & Dr. Makan Pourzandi	Thesis Supervisor

Approved by
Dr. Zachary Patterson, Graduate Program Director

October 27th, 2022
Dr. Mourad Debbabi, Dean
Gina Cody School of Engineering and Computer Science

Abstract

Provenance Analysis in Virtualized Environments

Azadeh Tabiban, Ph.D.
Concordia University, 2022

With the unprecedented need for remote working and virtual retail, there has been a worldwide surge in the adoption of cloud and edge computing. On the other hand, the significant reliance on virtual services has rendered the underlying virtualized environments supporting those services an attractive target for cyber criminals. There exist provenance-based solutions for identifying the root causes of security incidents and threat prevention by tracing the relationships between events at lower abstraction levels (e.g., system calls of an operating system). However, the sheer scale of virtualized environments means that such solutions would generate impractically large and complex provenance graphs for human analysts to interpret, especially in the context of virtualized environments with tens of thousands of users and inter-connected resources. Moreover, most intended user actions (e.g., creating a virtual function) generate a large number of events at lower abstraction levels, while it is typically challenging to associate those triggered operations to the intended actions of users, which further hinders understanding the provenance graphs. Finally, most works rely on human analysts to interpret provenance graphs into human-readable forensic reports.

Therefore, the main focus of this thesis is to facilitate the investigation and prevention of security incidents through practical provenance-based solutions in virtualized environments such as clouds and network functions virtualization (NFV). First, we propose a cloud management-level provenance model to facilitate forensic investigations by capturing the dependencies between cloud management operations, instead of low-level system calls. Based on this model, we design a framework to construct management-level provenance graphs and prune operations that are irrelevant to detected security incidents. Second, we propose an approach preventing security incidents in clouds based on the management-level provenance graph. Third, we propose the first multi-level provenance system for NFV built for capturing the relationship between management operations across different levels of the NFV stack, and increasing the interpretability of the logged information by leveraging the inherent cross-level dependencies. Fourth, we propose a solution to bridge the gap between human understanding of natural languages and data provenance by automatically generating forensic reports explaining the root cause of security incidents based on the provenance graphs.

Acknowledgments

My academic journey has been supported by many amazing people who have had a role in the outcome of this research one way or another. First, I would like to thank my academic supervisors, Professor Lingyu Wang and Professor Makan Pourzandi, for their guidance and support. I learned tremendously from them, and they are a great inspiration for me, not only in research, but also in terms of leadership and support. I have been repeatedly impressed by the extent they truly care about the success of their students. I will always be grateful and cherish how they believed in me and my potentials, which even helped myself to trust my abilities more.

This thesis benefited from my collaboration with Ericsson Research. I am particularly grateful for the wise and practical guidance I have received from my industrial advisors that I directly work with, including Dr. Makan Pourzandi and Dr. Yosr Jarraya. I also appreciate the insight and comments that I received from other specialists and researchers at different branches of Ericsson which contributed to building more practical and timely solutions in my research.

I also sincerely appreciate the support and advice from our Dean, Professor Mourad Debbabi, and thank my thesis committee for their insightful comments that improved the quality of my thesis and presentation. I am also thankful to my friends and lab mates for making our lab a welcoming home for me to pursue my Ph.D., and all the fruitful inspiring discussions, laughs and treats that we had together.

I would also like to express my deepest gratitude to my family. I appreciate how they encouraged my sense of creativity, taught me the value of self-reliance, and provided me with a great emotional security. I am a combination of who they are, and I feel I have achieved any success to this day with their mind and heart. I cannot be more grateful for their precious presence in my life and memory. Though, any word falls short in describing my appreciation.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Virtualized environments, such as clouds and network functions virtualization (NFV), have been widely adopted to provide users with the ability to self-provision their resources in a cost-effective manner over the shared physical infrastructure [114, 83]. On the other hand, the self-service and multi-tenancy nature of virtualized environments also implies a higher operational complexity and greater chances of misconfigurations [119, 21, 66]. Adversaries may exploit such misconfigurations to launch attacks on affected virtual resources. However, the sheer scale and complexity of those environments also render explaining what may have caused a security incident, i.e., *the root cause analysis*, far more challenging [1]. A manual approach to root cause analysis is typically impractical considering the sheer size of virtualized environments, and automated solutions become essential for understanding, debugging, and preventing security attacks exploiting either vulnerabilities or misconfigurations.

There exist root cause analysis solutions [2, 97, 1] that enable identifying failed components in virtualized environments. However, those solutions do not pinpoint the configuration changes causing the failures. On the other hand, provenance analysis (e.g., [49, 91, 92]) enables explaining system behaviors through tracing the interactions between events and data objects in a system.

However, adopting existing provenance-based solutions in the context of virtualized environments face certain challenges as we detail in Section 1.2.

1.2 Problem Statement

In this thesis, we address the limitation of existing solutions by facilitating provenance analysis in virtualized environments. In particular, this thesis seeks to answer the following questions:

- How can we facilitate the identification of root cause operations that led to security incidents in clouds and leverage the results to prevent recurring incidents?
- How can we facilitate the root cause analysis in multi-level NFV environments with potentially no obvious link between a detected incident and its root cause through impractically large and complex provenance graphs?
- How can we automate the interpretation of provenance analysis results into human-readable forensic reports?

We elaborate those problems as follows.

Challenging interpretation of provenance graphs in clouds. Most existing provenance-based solutions (e.g., [49, 92]) trace the relationships between system calls of operating systems at lower abstraction levels which would generate prohibitively huge provenance graphs, especially in the context of clouds with tens of thousands of cloud tenants and interconnected resources. Identifying the root cause operations that led to the attack is challenging using such provenance graphs. Additionally, the generated provenance graphs usually cannot be easily mapped to cloud management operations to understand what went wrong at that level.

Challenging multi-level root cause analysis in NFV environments. NFV enables agile deployment of network services on top of clouds. However, as NFV involves multiple levels of abstraction representing the same components, pinpointing the root cause of security incidents can become challenging. For instance, a security incident may be detected at a different level from where its root cause operations were conducted with no obvious link between the two. Moreover, existing provenance analysis techniques may produce results that are impractically large for human analysts to interpret due to the inherent complexity of NFV.

Costly manual report generation for security incidents. Most existing root cause analysis solutions rely on human analysts to interpret provenance graphs for root causes of security incidents. However, navigating and understanding a large and complex cloud-scale provenance graph can be very challenging for human analysts. Without such an understanding, cloud providers cannot effectively address the underlying security issues causing the incidents, such as vulnerabilities or misconfiguration.

1.3 Contributions

In this thesis, we build effective and practical provenance analysis solutions for investigating and preventing security incidents in virtualized environments. To this end, we design a cloud management-level provenance model to encode the interdependencies between management operations. We also propose a middleware-based framework to capture provenance metadata from different cloud services and construct the provenance graph for supporting forensic analysis and threat prevention. Next, we propose an interpretable multi-level provenance analysis approach for NFV environments. Our approach links the provenance graphs at different levels of the NFV stack and improves the interpretability of the results through removing irrelevant information and hiding redundant operations. Finally, we propose an approach to bridge the gap between data provenance and the results of analyses in the form of a natural language. Our approach leverages natural language translation techniques to automatically generate forensic reports explaining the root cause of security incidents based on the provenance graphs. We elaborate those contributions as follows.

1.3.1 Management-Level Forensic Analysis in Clouds

First, we propose a novel provenance-based solution that enables root cause identification at a higher abstraction level by tracing 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 operations, construct management-level provenance graphs and prune irrelevant operations. We implement our framework on OpenStack cloud 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 our solution incurs insignificant overhead and is scalable for clouds.

1.3.2 Preventing Recurring Security Incidents in Clouds

Second, we propose a management-level provenance solution to prevent security incidents in clouds through identifying the interdependent suspicious management operations. Specifically, after identifying the causally interdependent malicious management operations in clouds, our solution monitors the intercepted cloud management operations and blocks/notifies the identified chain of malicious operations to prevent the recurrence of the previously detected incidents. We implement this solution on OpenStack platform, and validate its effectiveness using security incidents based on real-world attacks. We also demonstrate that our approach incurs insignificant overhead and is scalable for clouds.

1.3.3 Towards Interpretable Multi-level Provenance Analysis in NFV

Third, we propose a provenance analysis system that handles the unique multi-level nature of NFV and assists the analyst to identify the root cause of security incidents. Specifically, we first define a multi-level provenance model to capture the dependencies between NFV levels. Next, we improve the interpretability through three novel techniques, i.e., multi-level pruning, mining-based aggregation, and rule-based natural language translation. We implement our approach on a Tacker-OpenStack NFV platform and validate its effectiveness based on real-world security incidents. We demonstrate that our solutions captures management API calls issued to all NFV services, and produces more interpretable results by significantly reducing the size of the provenance graphs (about 3.6 times reduction via the multi-level pruning scheme and two times reduction via the aggregation scheme). Our user studies show that our approach facilitates the analysis task of real-world users by generating more interpretable results.

1.3.4 Automatically Interpreting Provenance Graphs into Textual Reports

Fourth, we propose an automated approach for generating natural language forensic reports based on provenance graphs. Our main observation is that the way nodes and edges compose a path in provenance graphs is similar to how words compose a sentence in natural languages. Therefore, our solution leverages a novel combination of provenance analysis, natural language translation, and machine-learning techniques to generate forensic reports. We implement our solution on an OpenStack cloud testbed, and evaluate its performance based on real-world attacks. Our user study and experimental results demonstrate the effectiveness of our approach in generating high-quality reports (e.g., up to 0.68 BLEU score for precision).

The rest of this thesis is organized as follows.

- Chapter 2 provides a background on data provenance, cloud infrastructure model and NFV.
- Chapter 3 discusses the challenge of provenance analysis in cloud environments. We present our solution, *DominoBlocker*, which is a provenance-based framework focusing on management operations of cloud infrastructure. We explain our novel mechanisms for investigating and preventing security incidents in clouds using cloud management-level provenance analysis. Furthermore, we will detail the implementation and integration of our solution with OpenStack cloud platform.

- Chapter 4 discusses the challenge of root cause analysis in multi-level NFV environments. We then present our system, *ProvTalk*, for interpretable multi-level provenance analysis in NFV. We discuss our multi-level provenance construction mechanism, detail our three novel techniques to improve the interpretability of provenance graphs, and finally explain the implementation and evaluation results.

- Chapter 5 discusses the challenge of creating human-readable reports about the root cause of security incidents. We detail our framework, *VinciDecoder*, which automatically interprets provenance graphs into human-readable forensic reports. We detail our rule-based and learning-based mechanisms for generating forensic reports using provenance graphs. Moreover, we present our implementation details and evaluation results.

- Chapter 6 summarises our contributions in other co-authored publications focusing on cloud security auditing.

- Chapter 7 concludes the thesis with summarizing this research and discussing potential future directions.

Chapter 2

Background

This chapter provides a background on our research in data provenance, cloud infrastructure model and NFV.

2.1 Data Provenance

Data provenance is a powerful technique to trace the interactions between data objects (e.g., virtual resources or operating system files) and events (e.g., management operations or system calls) by capturing the dependencies between them in a graph representation, namely provenance graph. Based on a popular standard specification [18], nodes of a provenance graph are generally 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. Edges 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. In Chapter 3.3.2.1 and 4.4.1, we will explain how we leverage this specification to define our provenance model.

2.2 Cloud Infrastructure Model

Fig. 1 shows an example of a cloud virtual infrastructure (based on OpenStack concepts [123]), with cloud tenants provisioning and managing their virtual resources (e.g., VMs[1]) through management API interfaces (without loss of generality, our running example focuses on network-related security incidents).

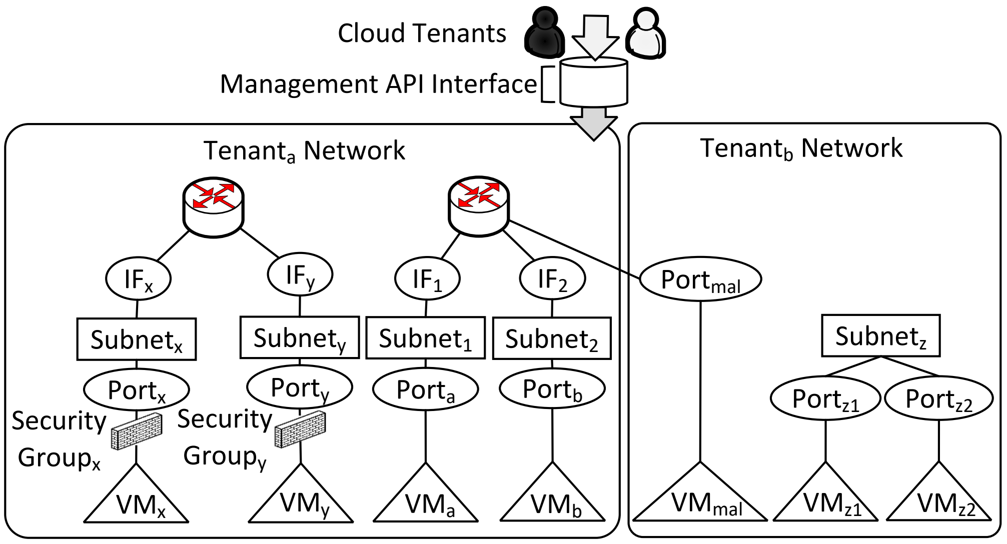


Figure 1: An example of cloud virtual infrastructure showing the dependencies between virtual resources.

Cloud Virtual Infrastructure. As shown in Fig. 1, in the cloud virtual infrastructure, routers interconnect subnets to route network traffic (e.g., between *Subnet$_1$* and *Subnet$_2$*). A subnet (e.g., *Subnet$_1$*) is associated with a Classless Inter-Domain Routing (CIDR), e.g., 10.0.0.0/24, and upon a tenant's request for the operation *Attach-Subnet-to-Router*, the subnet is attached to a router through an interface, e.g., the interface *IF$_1$*. Once a tenant requests for creating a VM, e.g., *VM$_a$*, the created VM is attached to a virtual port, e.g., *Port$_a$*. Ports can be created in subnets and each port is subsequently allocated with an IP address chosen from the IP address range of its connected subnet. Moreover, ports are attached with one or several security groups (i.e., arrays of network access rules specifying the allowed network traffic). Once a VM is attached to a security group, the

[1]The same concepts may be described by different terms in different platforms (e.g., *instance* or *server* [123] instead of VM).

IP table of that VM is updated with the IP addresses of VMs from/to which the network traffic is allowed according to the newly attached security group.

Interdependencies. Our above descriptions show that there may exist interdependencies between cloud virtual resources caused by management operations. For instance, the operation *Attach-Subnet-to-Router* causes an interdependency between a router and its attached subnets by adding the IP addresses of the subnets to the routing table of the router. Moreover, the operation *Add-Security-Group* causes an interdependency between the VMs attached to different security groups. In Section 3.3.2.1, we define our provenance model capturing such interdependencies.

2.3 NFV Background

The left side of Fig. 2 illustrates a high-level view of the ETSI NFV reference architecture, where a user-specified service description is implemented through the virtual network function (VNF) block (which provides a high-level representation of network functions) and the NFV infrastructure (NFVI) block (which represents the underlying cloud infrastructure), while both blocks are provided by the same network operator [31]. The right side of Fig. 2 shows an example of a multi-level network service deployment with corresponding management modules, and depicts how the actual deployment of a network service would correspond to the ETSI architecture. Specifically, the VNF block in the ETSI architecture maps to the *NFV level*, where a network service is deployed as several VNFs forming a VNF forwarding graph (VNFFG). The NFVI block in the ETSI architecture is mapped to both the service function chaining (*SFC level*), where there are virtual resources such as port pair groups, and the *cloud level*, where there are virtual resources such as VM, port and subnet. Finally, users can manage those levels through management modules including Network Function Virtualization Orchestrator (NFVO), Virtual Network Function Manager (VNFM), Software Defined Networking Controller (SDN-C) and Virtualized Infrastructure Manager (VIM).

Fig. 2 demonstrates how dependencies may exist between resources either at the same level or between different levels. For example, the solid lines at the NFV level indicate that the two VNFs,

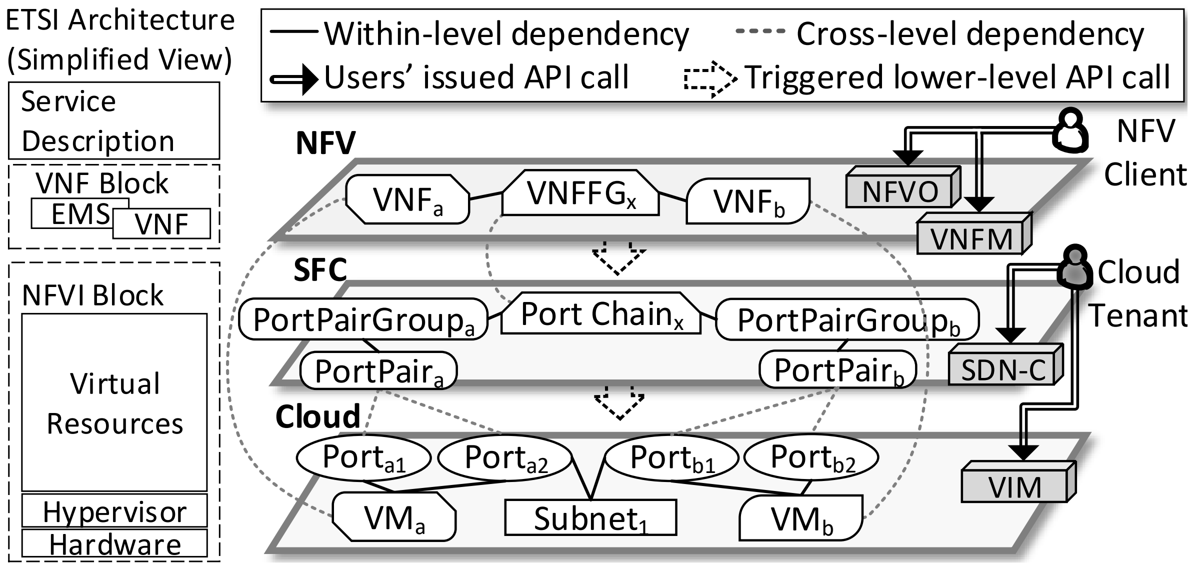


Figure 2: High-level view of ETSI NFV reference architecture (left); Example of multi-level network service deployment showing the dependencies between resources (right).

namely VNF_a and VNF_b, are part of (and connected through) a VNFFG named $VNFFG_x$ (similar dependencies exist between the port pair groups and port chains, the VMs and ports, as well as the ports and subnets). Furthermore, the dashed lines represent dependencies across different levels. For example, the dashed line between VNF_a (NFV-level) and VM_a (cloud-level) indicates that the creation of VNF_a will automatically trigger the creation of VM_a. Similarly, the creation of $VNFFG_x$ will automatically trigger the creation of $PortChain_x$. Finally, a dashed line links $PortPair_a$ (SFC level) to its two components ($Port_{a1}$ and $Port_{a2}$).

The above example shows how operations executed at different levels might affect resources and consequently introduce (within-level or cross-level) dependencies in the NFV stack. These dependencies are crucial to correctly identify the root cause of security incidents in NFV. To capture those concepts, we define a multi-level provenance model in Section 4.4.1.

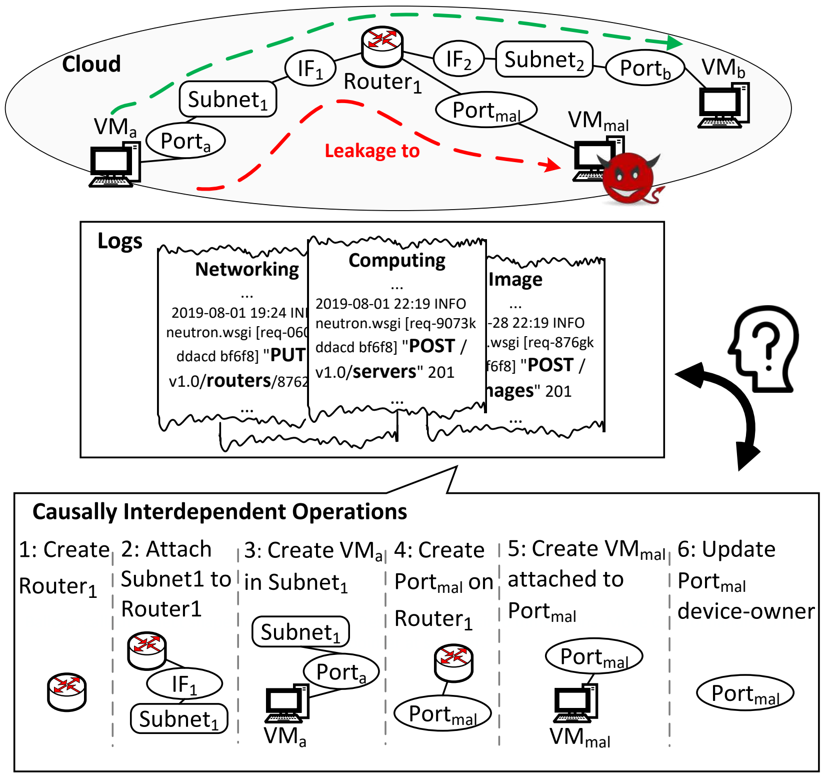
Chapter 3

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

3.1 Introduction

Cloud computing provides the users with the benefit of provisioning their resources over the optimally shared underlying physical infrastructure. However, the self-service and multi-tenancy nature of clouds also leads to a higher complexity and greater chances of misconfigurations [119, 21, 66]. Adversaries may exploit those misconfigurations to launch attacks on virtual resources.

The added complexity of cloud environments may also render the important task of pinpointing the root cause of security incidents far more challenging [1]. There exist root cause analysis solutions [2, 97, 1] for identifying failed components (e.g., switches) 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 [117, 92, 91, 49], they become insufficient in the context of clouds, as such solutions would generate a prohibitive amount of provenance metadata without providing a big picture about the root cause.


Cloud
VM_a
Port_a
Subnet_1
IF_1
Router_1
IF_2
Subnet_2
Port_b
VM_b
Port_mal
VM_mal
Leakage to
Logs
Networking
Computing
Image
Causally Interdependent Operations
1: Create Router_1
2: Attach Subnet1 to Router1
3: Create VM_a in Subnet_1
4: Create Port_mal on Router_1
5: Create VM_mal attached to Port_mal
6: Update Port_mal device-owner

```
...
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 /
nages" 201
...
```

Figure 3: An example of data leakage in clouds (top), logs of various cloud services (middle), and the interdependent management operations leading to the attack (bottom).

Motivating Example. Fig. 3 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 cloud services (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 (*Port*$_{mal}$) 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

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

(`OSSA-2015-018`[1]) 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 $Subnet_1$, $Router_1$ and $Subnet_2$.

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 creation of VM_a and the attachment of VM_{mal} to $Port_{mal}$. 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.

To address those challenges, we propose in this research *DominoBlocker*, a provenance-based solution for easier forensic analysis and prevention of security incidents in clouds. Our key idea is to lift the provenance analysis to the cloud management-level, which enables tracing configuration changes and identifying the ones causing attacks. Specifically, we first define a provenance model to encode the interdependencies between management operations, virtual resources and inputs in clouds. Moreover, we propose a middleware-based framework to intercept provenance metadata from management API calls made to different cloud services and construct the provenance graph. Additionally, we devise provenance-based forensic and prevention mechanisms, which leverage cloud-specific user-oriented pruning and label propagation techniques, respectively. Finally, we implement and experimentally evaluate a prototype of DominoBlocker 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-based 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

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

big picture about cloud configuration changes with increased interpretability that facilitates subsequent analyses.

- In lifting the provenance analysis to the management-level, a set of novel mechanisms are proposed as follows. First, our middleware-based solution can enable incremental provenance construction, runtime analysis and enforcement by intercepting management API calls issued to cloud services. Moreover, our user-oriented pruning techniques allow cloud tenants to customize their forensic analysis based on their needs, security assumptions and user preferences. Furthermore, our threat prevention mechanisms enable tenants to specify shortlisted operations that trigger the verification of monitoring policies. Our label-based threat prevention approach enables easier specification and more efficient verification of monitoring policies.

- Our evaluation results show that DominoBlocker can provide a scalable tool for identifying the root cause and preventing security incidents in cloud infrastructures with insignificant performance and storage overhead.

3.2 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 [57, 101], hash-chain-based provenance storage protection [41] or type enforcement [16].

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.

3.3 Methodology

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

3.3.1 Overview

Fig. 4 depicts all three major modules of DominoBlocker and an overview of their mechanism. Our runtime provenance construction module (solid line arrows in Fig. 4) incrementally constructs the provenance graph based on the intercepted management API calls. Once a security incident is detected (e.g., by a deployed intrusion detection system), our offline forensics analysis module (dashed line arrows in Fig. 4) assists the analyst to identify the chain of suspicious operations by pruning the provenance graph while causing no additional runtime delay. To prevent security incidents, our runtime threat prevention module (dotted line arrows in Fig. 4) monitors the incrementally constructed provenance graph and notifies or blocks the identified suspicious operations captured by the provenance graph before they are applied to the cloud. We discuss these modules in detail as follows.

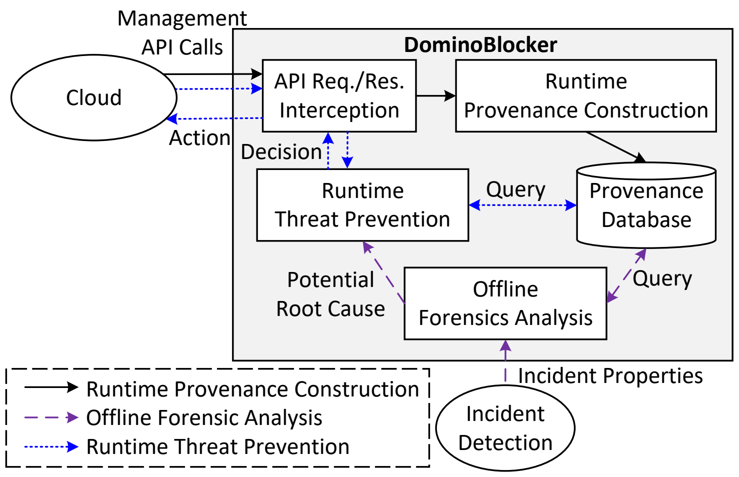

Management API Calls
DominoBlocker
Cloud
API Req./Res. Interception
Runtime Provenance Construction
Action
Decision
Runtime Threat Prevention
Query
Provenance Database
Potential Root Cause
Offline Forensics Analysis
Query
Incident Properties
Incident Detection
Runtime Provenance Construction
Offline Forensic Analysis
Runtime Threat Prevention

Figure 4: A highlevel overview of DominoBlocker.

3.3.2 Runtime Provenance Construction

In this section, we define our cloud management-level provenance model and describe the provenance construction module.

3.3.2.1 Cloud Management-Level Provenance Model

In general, *provenance* usually refers to a technique that captures the information flow between sources and sinks [49]. In the context of cloud virtual infrastructures, we identify as sinks the management operations (e.g., *CreateVM* and *Updateport*) 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 [18]. 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

Table 1: Mapping the common concepts in clouds to the PROV-DM Model.

Cloud Concept	**Description**	**PROV Model**	**Subtype**
Cloud Tenant	Groups of users owning an isolated set of virtual resources.	Agent	Tenant Admin, other tenants
Cloud User	Customers 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, 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 states of a virtual infrastructure configuration, e.g., configuration state of the virtual hardware for VMs.	Entity	Security groups, Flavors, etc.
Input changing configurations	The input data causing a change to the configuration state.	Entity	Security group rules, etc.

cloud virtual infrastructure management to this specification. Specifically, a summary of our provenance model is shown in Table 1. Subtypes are added to refine the classification of PROV-DM based on our needs. To illustrate our model, Fig. 5 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 8, entity vertices (shown as ovals in Fig. 5) 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*. To avoid cycles, we adopt the node versioning method as in [75, 91, 92], by creating a new version of an entity once an operation affects its represented resource[1]. For example, as it is shown in Fig. 5, a new node representing

[1]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 5.4.

an updated version of *Router$_1$* (i.e., the node ⟨*Router$_1$*, *Version1*⟩) is created when it is attached to *Subnet$_1$*. 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 the represented resources, the time at which the corresponding management API call is intercepted, 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 Fig. 5) represent management API calls made to either change the state of resources (e.g., *StartVM*) or to mange their lifecycle (e.g., *CreateVM*). 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., *CreateVM*. 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 Fig. 5) 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.

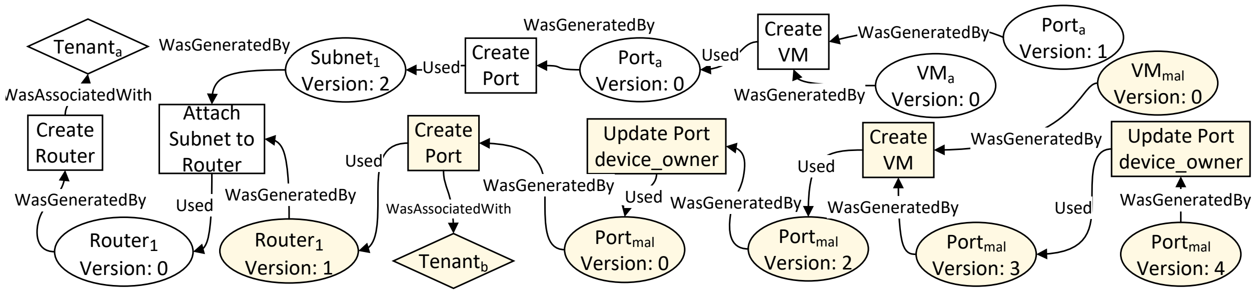


Figure 5: The provenance graph corresponding to the scenario of Fig. 3. Nodes related to the attacker's steps are highlighted in yellow.

3.3.2.2 Building the Provenance Graph

Based on our defined cloud management-level provenance model, DominoBlocker constructs the provenance graph during runtime in two main steps: data collection and graph generation.

Data Collection. To capture all management operations affecting cloud virtual resources, we design and deploy our interception mechanism as middlewares [107, 60] attached to all cloud managerial services. These services include networking, computing, and storage services. Note that this approach allows us to tackle the challenge of capturing all information required for provenance construction, while standard cloud logs generally lack some details about the requested operations [67] (e.g., operations *StartVM* and *StopVM* may appear to be identical in the logs). Moreover, DominoBlocker processes the intercepted management API calls according to the analyst's pre-specified rules. These rules are specified based on cloud API documentations [3] and applied to extract the type of requested operations, the affected resources, and the user requesting the operation. More details on parsing the intercepted API calls and retrieving their parameters will be provided in Section 3.5.

Graph Generation. At this step, DominoBlocker 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 node for each affected virtual resource and a node for the requested operation. Next, it creates edges to capture the dependencies between operations and the affected resources: *Used* edges are created from the operation node to the nodes representing the previous version of the affected resources. *WasGeneratedBy* edges are created from the newly created resource nodes to the created operation node. Moreover, DominoBlocker creates a *WasAssociatedWith* edge from the operation node to the node representing the cloud user/tenant requesting those operations.

Example 1. According to Fig. 5, DominoBlocker creates a *WasAssociatedWith* edge from the *CreateRouter* operation node to the node representing $Tenant_a$ (i.e., the cloud tenant created that router). Upon the interception of the operation *Attach-Subnet-to-Router*, DominoBlocker creates a node representing this operation. DominoBlocker also creates the node $\langle$*Router$_1$*, *Version1*$\rangle$ and a *WasGeneratedBy* edge from this node to the node *Attach-Subnet-to-Router*. It also creates a *Used* edge from the node *Attach-Subnet-to-Router* to the node $\langle$*Router$_1$*, *Version0*$\rangle$, which represents the previous state of *Router$_1$*.

In Section 3.3.3 and 3.4, we detail how the constructed provenance graph is leveraged to identify and prevent the suspicious operations leading to an incident, respectively.

3.3.3 Offline Forensics Analysis

This module assists the analyst to identify the management operations leading to detected security incidents using the provenance graph constructed prior to the detection. To detect the incident, we can leverage existing detection mechanisms monitoring the infrastructure and the deployed VMs. For instance, we can rely on three main types of detection methods: VM monitoring techniques [17] (e.g., intrusion detection tools), cloud policy compliance verification [4], and cross-layer consistency verification tools [63].

The analyst may face two challenges in identifying the root cause operations based on the provenance graph. First, the provenance graph may include many irrelevant operations, which may be challenging for the analyst to distinguish from the root cause operations. Second, the multi-tenancy nature of clouds implies that different tenants may have significantly different requirements based on their security assumptions and objectives of the investigation. To address these issues, we propose two user-oriented pruning mechanisms, namely *disjoint subgraph* pruning and *context-based* pruning, to automatically identify and remove irrelevant information from the provenance graph. The analysts could narrow down the scope of their investigation by triggering the pruning process and selecting the proper pruning mechanisms based on their requirements.

Disjoint Subgraph Pruning. This pruning mechanism removes all nodes that are not connected to the subgraph attached to the node representing the target resource (i.e., the resource associated with the detected incident). To this end, DominoBlocker first identifies the potentially relevant nodes by

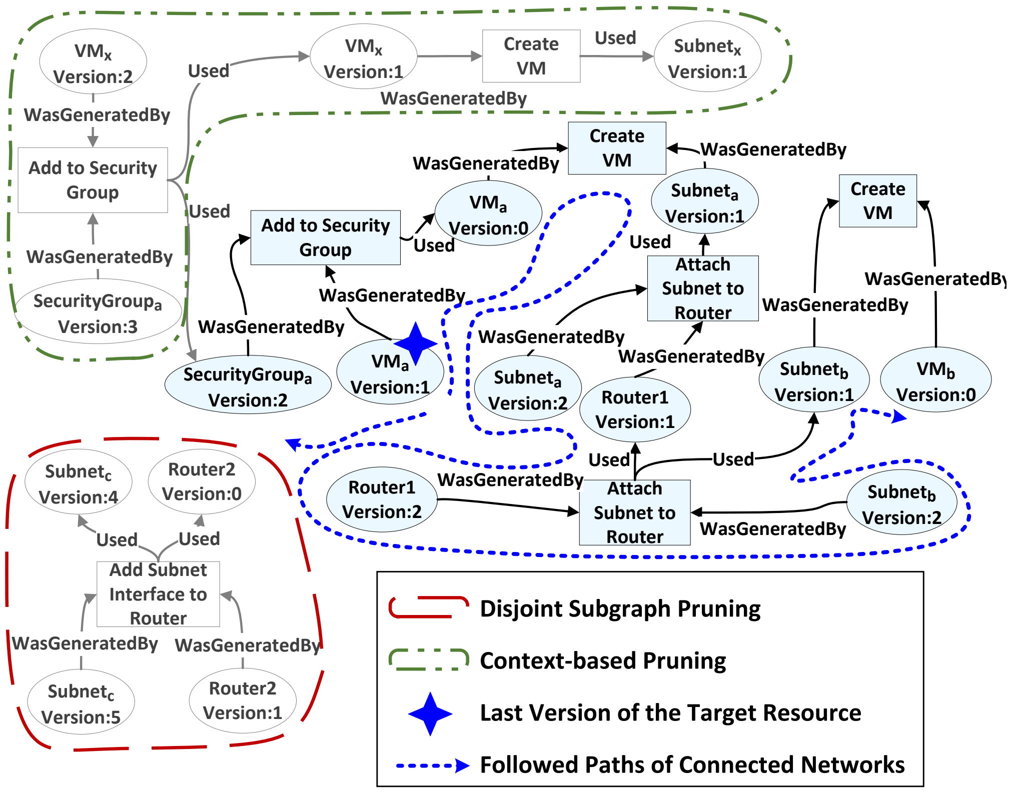


Figure 6: Example of pruning through finding the provenance graph nodes disjointed from the target VM (e.g., VM_a) and unrelated nodes according to the security incident context.

following all paths of the form *Resource1 - (Used/WasGeneratedBy) - ManagementOperation - (WasGeneratedBy/Used) - Resource2* starting from the last version of the target resource node. Next, DominoBlocker removes all nodes that cannot be reached through following such paths, as the absence of a path between a group of nodes and the target resource implies the lack of interdependencies between them and thus the former may be pruned.

Example 2. Fig. 6 shows an example provenance graph after applying disjoint subgraph pruning. DominoBlocker starts following the paths from the starred VM_a node and reaches the *Add-Security-Group* node, which *used* the previous version of VM_a and *generated* the new version of $SecurityGroup_a$. DominoBlocker further follows the operations affected VM_a earlier in the provenance graph, and reaches the $Subnet_a$ node through the *WasGeneratedBy* edge from the *CreateVM* node. It also reaches the *Attach-Subnet-to-Router* node that *used* $Subnet_a$ and *generated* a new version of *Router1*. Following this process, DominoBlocker finds all operations and resources that potentially affected the target resource, VM_a, as well as the operations and resources which are later impacted by the changes made to VM_a. On the other hand, the disjoint subgraph (e.g., $Subnet_c$, *Router2*, and the operations affecting them), shown inside the dashed contour in red, is pruned since there is no dependency between those nodes and VM_a.

Context-Based Pruning. This pruning mechanism removes nodes that are not contextually dependent on the target resource according to the analyst's provided criteria. The analyst may specify those criteria based on the security assumptions in the cloud platform or detected incidents. DominoBlocker follows paths in the provenance graph while checking the specified constraints to identify a subgraph of resources and operations interdependent with the target virtual resource, and remove the remaining nodes and edges.

Example 3. Fig. 6 shows an example of context-based pruning in the context of VM_a data leakage incident. Since the incident is detected by a network-based IDS, the analyst attempts to investigate only the nodes of resources that are in a direct network connectivity with VM_a as well as its attached network, $Subnet_a$, and prune other nodes from the provenance graph. Based on a pre-specified set of operations (e.g., *CreateVM* and *Attach-Subnet-to-Router*) that potentially create or update network connections between resources, DominoBlocker automatically identifies resources connected to $Subnet_a$ through the provenance graph. To this end, DominoBlocker starts from the last version of VM_a node in the provenance graph and follows paths to identify the potentially relevant nodes until it reaches operations that are not in the specified set. For instance, DominoBlocker keeps VM_b and its attached subnet, $Subnet_b$, in the provenance graph as they are reachable from VM_a through *Attach-Subnet-to-Router* and *CreateVM* nodes. On the other hand, VM_x, connected to $Subnet_x$ through *CreateVM* node (in the green contour), is pruned in this step, as it cannot be reached from VM_a through the pre-specified operations.

Our offline forensics analysis module allows a more focused investigation towards identifying the potential root cause. In the next section, we detail how DominoBlocker leverages the results of forensics analysis to prevent the recurrence of security incidents in clouds.

3.4 Threat Prevention

To prevent the recurrence of security incidents, we propose two different mechanisms: *regular* threat prevention and *label-based* threat prevention.

3.4.1 Regular Threat Prevention

Fig. 7 shows the main steps of our regular threat prevention scheme leveraging the output of our forensics analysis module: 1) DominoBlocker either leverages an existing solution [36] or an analyst to specify a monitoring policy indicating the graph representation of the interdependent suspicious operations (i.e., the operations identified through our forensic analysis). 2) During runtime, DominoBlocker verifies the specified monitoring policy against the provenance graph updated upon intercepting new management API calls. 3) DominoBlocker notifies/blocks the intercepted management API calls if it discovers the chain of suspicious operations is captured by the updated provenance graph. 4) Additionally, DominoBlocker allows the analyst to further examine the operations captured prior to previous incidents, and 5) refine the specified chain of suspicious operations accordingly. In Section 3.4.2, we explain how we modify those steps to support our proposed label-based threat prevention approach.

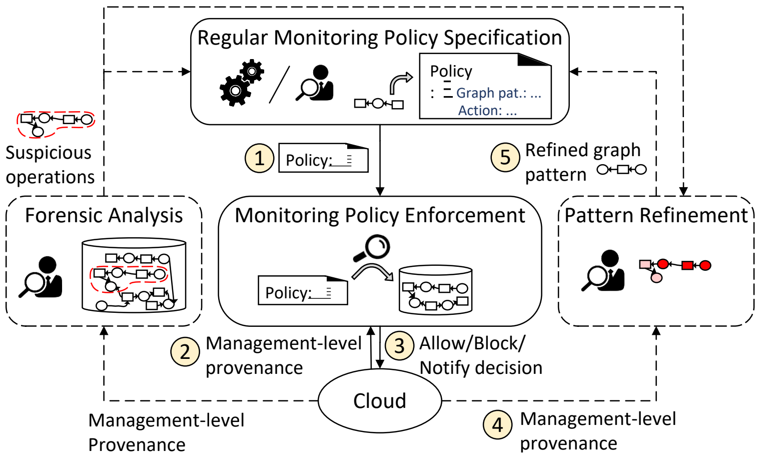

Regular Monitoring Policy Specification
Policy
Graph pat.: ...
Action: ...
Suspicious operations
1 Policy:
5 Refined graph pattern
Forensic Analysis
Monitoring Policy Enforcement
Policy:
Pattern Refinement
2 Management-level provenance
3 Allow/Block/ Notify decision
Cloud
Management-level Provenance
4 Management-level provenance

Figure 7: The steps of our regular threat prevention scheme. The dashed arrows and blocks correspond to the optional steps.

3.4.1.1 Regular Monitoring Policy Specification

To prevent the recurrence of incidents, DominoBlocker is provided with monitoring policies consisting of a *suspicious graph pattern* (a graph query capturing the suspicious operations) and an *enforcement* action (e.g., notify or block). To obtain the graph pattern, DominoBlocker may leverage an existing work [36] to automatically obtain the graph patterns based on the results of forensic investigations. Optionally (e.g., to ensure the correctness of the specified patterns), DominoBlocker allows the analyst to manually specify the graph patterns. The specified graph pattern may correspond to one of the following cases: 1) the precise attack steps: the operations precisely identified through forensics investigations, 2) imprecise attack steps: the attack steps as well as several irrelevant operations (i.e., the operations that could not be differentiated from the attack steps by the analyst through the forensics investigation, and thus are added to the suspicious graph pattern), and 3) cloud security policies: a chain of operations that may violate the security requirements (e.g., network isolation between VMs) of the cloud platform. Such properties are determined based on the cloud tenants' preferences and may be independent from the result of forensics analyses. The specified monitoring policy is fed into DominoBlocker (Step 1 in Fig. 7) for preventing future security incidents.

Example 1. Fig. 8 illustrates a monitoring policy preventing the attack described in our Motivating Example (Section 5.1, Fig. 3). *Pattern* shows the graph representation of the suspicious operations described in a graph query language (*Cypher*[1]). This monitoring policy specifies DominoBlocker to notify the cloud admin once it detects the graph representation of operations creating a port on another tenant's router, or updating the *device_owner* field of a port immediately after that port is attached to a newly created VM.

[1]Cypher Language, `https://neo4j.com/developer/cypher-query-language`

```
Pattern: {
//to detect disabling anti-spoofing rules
MATCH (op1:'CreateVM')<-[]-(r1:'port')<-[*]-(r2:'port')<-[]-(op2:'UpdatePortDeviceOwner'}
WHERE op2.TimeInSecond-op1.TimeInSecond < 1 AND r1.ID=r2.ID
//to detect accessing other tenants' network
OR ((resource:'router')<--(op3:'CreatePort') AND resource.TenantID<>op3.TenantID)
RETURN op1.RequestID, op3.RequestID}
//to notify the admin about the potential threat
Action: notify
```

Figure 8: The regular monitoring policy used to prevent the incident in our Motivating Example (Section 5.1).

3.4.1.2 Monitoring Policy Enforcement

This module verifies the provenance graph against the specified monitoring policy during runtime. Specifically, at the interception of every management API call and before its execution, DominoBlocker updates the provenance graph (as explained in Section 3.3.2.2). Next, DominoBlocker verifies whether the suspicious graph pattern (e.g., Pattern in Fig. 8) is detected in the provenance graph updated by the newly intercepted API call (Step 2 in Fig. 7). To this end, DominoBlocker issues a query to the graph database searching for the suspicious graph pattern in the provenance graph. If the graph pattern is detected, DominoBlocker either notifies the cloud admin about the potential threat or blocks the newly requested operation (Step 3 in Fig. 7) based on the enforcement action specified by the monitoring policy. To block a requested operation, DominoBlocker discards the intercepted management API call and generates an error message displayed to the cloud user. Moreover, DominoBlocker deletes the nodes and dependencies corresponding to the blocked operation from the updated provenance graph. To avoid unnecessary delays, DominoBlocker can be configured to trigger the verification upon the interception of a subset of API calls specified by the analyst.

The analyst can configure DominoBlocker to conduct the enforcement once a portion of the suspicious graph pattern is detected in the provenance graph. This is especially useful for cases where the root cause operations are not precisely identified (the case of imprecise attack steps in Section 3.4.1.1), and thus, the graph pattern includes operations irrelevant to the attack. Hence, by enforcing the monitoring policy upon detecting a portion of the graph pattern, it is more likely to prevent the completion of the attack. DominoBlocker also allows the analyst to periodically refine the suspicious graph pattern. To this end, the analyst monitors the provenance graph constructed prior to the incidents that could not be prevented by the specified monitoring policy (Step 4 in Fig. 7). Since the irrelevant operations may occur less frequently prior to the recurrences of the same incident, the analyst refines the specified graph pattern by removing such operations and provides the updated monitoring policy (Step 5 in Fig. 7).

3.4.2 Label-based Threat Prevention

To facilitate the specification and verification of monitoring policies, we propose a label-based threat prevention approach based on our regular prevention mechanism. In the following, we detail our motivation for label-based threat prevention, its mechanism, and benefit over our regular threat prevention mechanism.

Motivation. Specifying regular monitoring policies involves formalizing interdependent suspicious operations into detailed graph patterns, which may require extensive manual efforts. Moreover, the analyst may need to specify several graph patterns corresponding to different scenarios leading to similar security incidents, which increases the required effort. Finally, attempting to detect suspicious graph patterns may trigger time-consuming queries for following paths of the provenance graph. Hence, we propose a label-based threat prevention mechanism that enables specifying more generic graph patterns covering multiple scenarios leading to the same incident, and allows the enforcement of monitoring policies through single dependency verification instead of potentially time-consuming queries following long paths of the provenance graph.

Fig. 9 shows the five major steps of our label-based threat prevention approach designed based on our regular threat prevention scheme. In the following, we detail these steps.

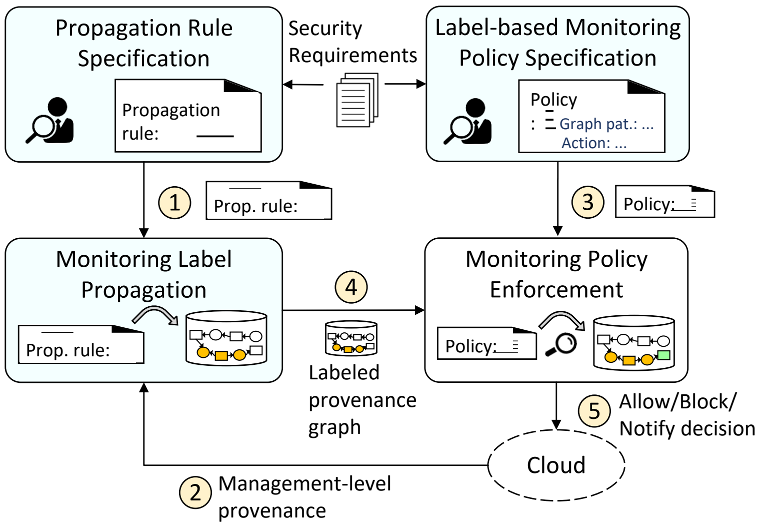

Propagation Rule Specification
Propagation rule:
Security Requirements
Label-based Monitoring Policy Specification
Policy
Graph pat.: ...
Action: ...
1
Prop. rule:
3
Policy:
Monitoring Label Propagation
Prop. rule:
4
Labeled provenance graph
Monitoring Policy Enforcement
Policy:
5
Allow/Block/ Notify decision
Cloud
2
Management-level provenance

Figure 9: Steps of our label-based prevention approach designed based on our regular threat prevention mechanism (Fig. 7). The colored boxes are specific to our label-based prevention.

3.4.2.1 Propagation Rule Specification

To track changes affecting security-sensitive resources, DominoBlocker allows the analyst to assign monitoring labels to different resource nodes, and specify label propagation rules (Step 1 in Fig. 9). Specifically, cloud admins may have different assumptions about the security sensitivity of resources based on the data processed by those resources, users accessing them, etc. Accordingly, the analyst assigns monitoring labels to resources representing their different levels of sensitivity. For instance, the nodes of VMs processing sensitive or non-sensitive information are assigned with monitoring labels *Sensitive* and *Nonsensitive*, respectively. Moreover, the analyst provides propagation rules that specify the properties of nodes that may pass the monitoring labels to their adjacent nodes.

3.4.2.2 Monitoring Label Propagation

During runtime, monitoring labels are stored as a node property upon the creation of the resource and its corresponding node. Next, DominoBlocker iteratively passes the monitoring label to the nodes with properties specified by the propagation rules that are adjacent to the labeled nodes (Step 2 in Fig. 9).

Example 2. Fig. 10 (left) shows an example where a cloud admin requires network isolation between VM_a processing sensitive information and VM_b. In Fig. 10 (right), we can see two scenarios (i.e., Scenario A and Scenario B) leading to direct network connectivity between VM_a and VM_b. To enable label-based threat prevention, the nodes representing VM_a and VM_b are assigned with monitoring labels *Sensitive* and *Nonsensitive* upon the creation of those VMs. To identify all resources in network connectivity with VM_a and VM_b, Fig. 11 shows an excerpt of the propagation rules (in Cypher language) that passes the monitoring labels to the nodes representing connected resources through the operations *CreateVM* and *Attach-Subnet-to-Router*.

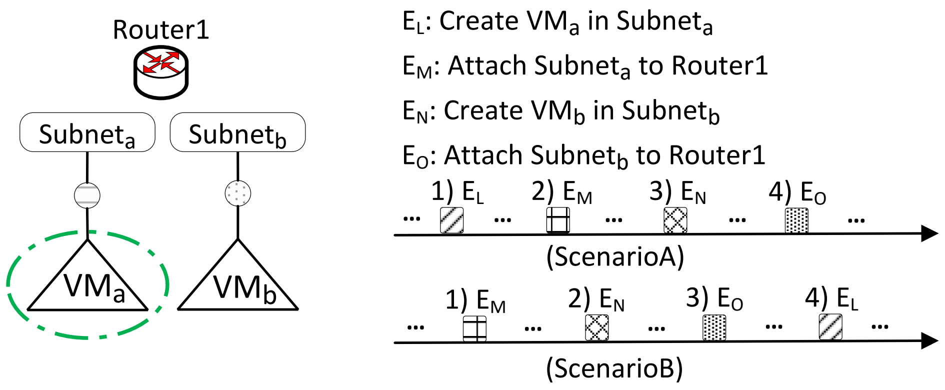

Router1
Subnet$_a$
Subnet$_b$
VM$_a$
VM$_b$
E$_L$: Create VM$_a$ in Subnet$_a$
E$_M$: Attach Subnet$_a$ to Router1
E$_N$: Create VM$_b$ in Subnet$_b$
E$_O$: Attach Subnet$_b$ to Router1
1) E$_L$ 2) E$_M$ 3) E$_N$ 4) E$_O$
(ScenarioA)
1) E$_M$ 2) E$_N$ 3) E$_O$ 4) E$_L$
(ScenarioB)

Figure 10: Examples of two scenarios leading to network connectivity between VM_a and VM_b.

3.4.2.3 Label-based Monitoring Policy Specification and Enforcement

To prevent resources with different levels of sensitivity from affecting each other, the label-based monitoring policy specifies to notify/block the operations that create a dependency between nodes with different monitoring labels (Step 3 in Fig. 9). During runtime, DominoBlocker verifies the specified monitoring policy against the labeled provenance graph (Step 4 in Fig. 9). To enforce the policy, DominoBlocker evaluates the monitoring labels of the nodes representing the most recent version of the affected resources (Step 5 in Fig. 9). We note that this is in contrast with our regular threat prevention mechanism, which may require tracing back the created operation node to the initial versions of its affected resources on potentially long paths of the provenance graph.

```
while (srcRes1s):
   srcRes1 = srcRes1s[0]
   query = "WITH  %s AS srcRes1s\
           OPTIONAL MATCH (res1{uuid:'%s'})-[:WasGeneratedBy]->(op) <-
           [:WasGeneratedBy]-(srcres2) \
           WHERE res1.label<>[] AND srcres2.label=[] AND op.eventName IN
           ['CreateVM','AttachSunetToRouter'] \
           \\To propagate labels through WasGeneratedBy edges and the listed
           operations to other resources if they are not labeled
           SET op.label = res1.label \
           \\For example, in Figure 7c, <SubnetB, version1> would be labeled
           SET srcres2.label = res1.label \
           ...
           \\The same matches for Used edges (which label e.g., <Router1, version1>
           ...
           \\To continue propagation:
           WITH  srcres2, srcres22, FILTER(x IN srcRes1s \
           WHERE x <> srcRes1s[0]) AS srcRes1s \
           WITH srcRes1s + COLLECT (srcres2.uuid) AS srcRes1s \
           RETURN srcRes1s  " %(srcRes1s, srcRes1, srcRes1, srcRes1)
   srcRes1s = tx.run(query)
   srcRes1s = srcRes1s.data()[0].get('srcRes1s')
```

Figure 11: An example of propagation rules labeling nodes representing resources with network connectivity to a VM.

Example 3. Fig. 12(a) depicts a label-based monitoring policy preventing the scenarios in Fig. 10 (Example 2) by blocking the operations causing dependencies between resources with different monitoring labels. Fig. 12(b) and 12(c) show the provenance graphs corresponding to Scenario A and Scenario B of Example 2, respectively, that are labeled based on the rules in Fig. 11. In Scenario A (Fig. 12(b)), once a user creates VM_a, DominoBlocker assigns the monitoring label *Sensitive* to the node representing VM_a and propagates this label to its connected network ($Subnet_a$) through the *CreateVM* operation node (Step 1 in Fig. 12(b)). After intercepting the operation *Attach-Subnet-to-Router*, DominoBlocker propagates the monitoring label *Sensitive* to the $Router_1$ resource node through the node representing the newly intercepted operation (Step 2 in Fig. 12(b)). Similarly, $Subnet_b$ node is assigned with the monitoring label *Nonsensitive* due to its connection with VM_b (Step 3 in Fig. 12(b)). Once a user attempts to enable network connectivity between VM_a and VM_b by attaching $Subnet_b$ to $Router_1$ (Step 4 in Fig. 12(b)), DominoBlocker verifies only the label of the most recent version of $Subnet_b$ and $Router_1$, and blocks this operation according to the specified monitoring policy (Fig. 12(a)). Likewise, in Fig. 12(c), DominnoBlocker automatically propagates the monitoring label *Nonsensitive* to networks and resources with network connectivity to VM_b for preventing Scenario B. Subsequently, the creation of VM_a in $Subnet_a$, and its subsequent network connectivity with VM_b is blocked as $Subnet_a$ is assigned with the monitoring label *Nonsensitive* due to its network connection with $Subnet_b$ and VM_b. As we can see, our label-based mechanism prevents both scenarios without following the potentially long paths at the interception of operations breaching the network isolation.

We conclude that our label-based threat prevention approach enables easier specification and monitoring of policies. Note that although we need our regular monitoring policies for monitoring the specific properties of nodes (e.g., the elapsed time between operation nodes in our Motivating Example), our label-based prevention mechanism can significantly ease the prevention of a vast group of incidents where the attackers attempt to affect resources with different sensitivity levels. In Section 3.6, we demonstrate that this approach also reduces the delay, especially in monitoring larger provenance graphs.

3.5 Implementation

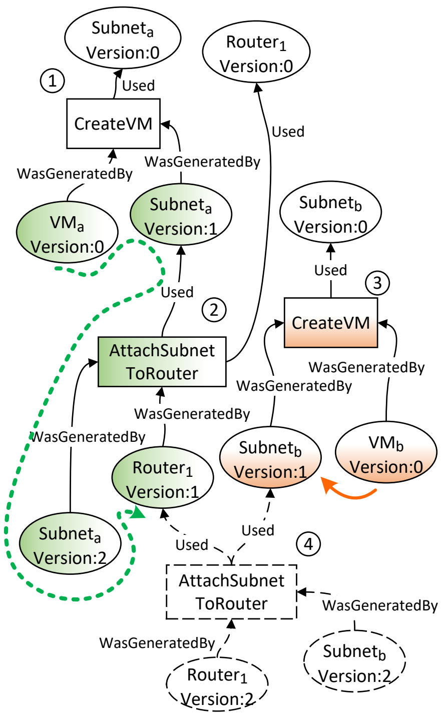
We implement DominoBlocker in a testbed based on OpenStack (a popular open source popular cloud platform [123]). We note that only our data collection and enforcement mechanisms are platform-specific, while the modular design of DominoBlocker makes it easily adaptable to other cloud platforms (detailed discussion in Section 3.7). In the following, we describe the integration of DominoBlocker with OpenStack and the architecuture of our approach as shown in Fig. 13.

Integration with OpenStack. We deploy DominoBlocker as a middleware and conduct a set of preprocessing to enable intercepting management API calls and obtaining provenance metadata:

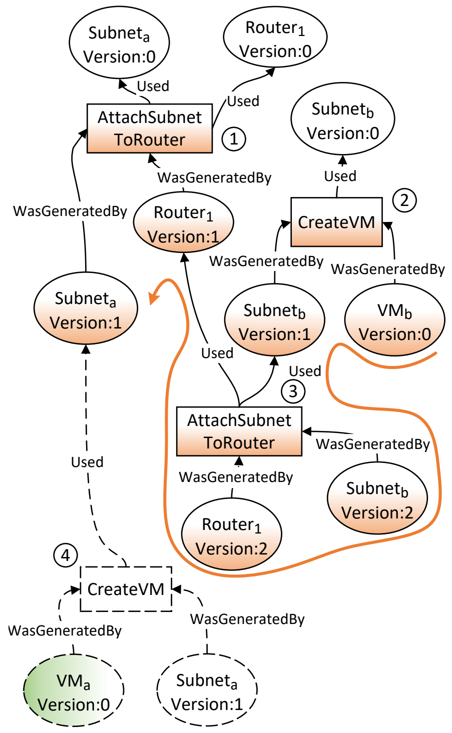
1. *Intercepting Management API Calls:* To intercept REST API calls issued to OpenStack services (e.g., Nova and Neutron), we implement our framework as a Python *WSGI* middleware

```
Pattern: {
MATCH (a)--(operation)--(b)
WHERE a.Label < > b.Label and operation.RequestID = $InterceptedRequestID
RETURN operation.RequestName, operation.TenantID}
Action: block
```

(a) The label-based monitoring policy preventing the direct network connectivity between VM_a and VM_b.



(b) Provenance graph, Scenario A.



(c) Provenance graph, Scenario B.

Figure 12: Leveraging label-based monitoring policies to prevent scenarios of Fig. 10. Dashed nodes and edges are related to the blocked operations. Green and orange nodes are assigned with *Sensitive* and *Nonsensitve* monitoring labels, respectively.

similar to existing works [107, 60], and insert it into the filter chain ending at those services. Fig. 14(a) depicts an excerpt of the updated Neutron API configuration and its filter chain into which DominoBlocker is inserted. This configuration is stored in the *api-paste.ini* file corresponding to each service.

2. *Preprocessing for Information Extraction:* To obtain provenance metadata, we specify parsing and operation typing rules, based on which DominoBlocker will extract the information

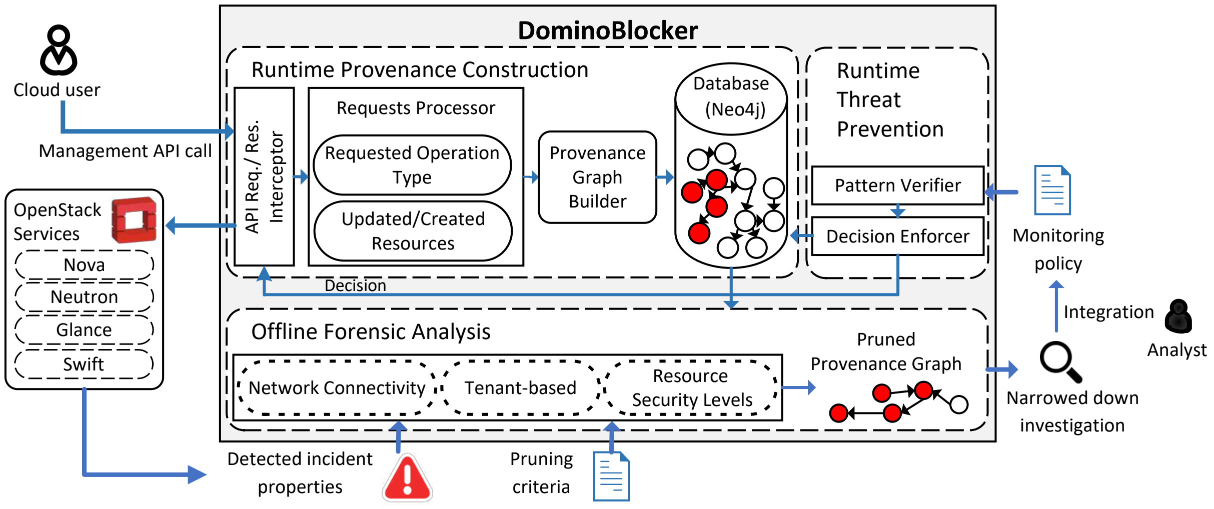


Figure 13: The architecture of DominoBlocker and its integration with OpenStack.

of the intercepted management API calls, and identifies the type of the requested operations. DominoBlocker allows users to focus provenance analysis on a set of operations and resources as well as OpenStack services. For instance, to investigate and prevent network-related incidents, the analyst may provide parsing and typing rules only for management API calls that are sent to Neutron service (networking service of OpenStack) and may affect network-related configurations. Fig. 14(b) shows selected fields of an example API call attaching a port to a VM. In this example, the ID of the attached port (*f91398*) can be extracted by parsing the *PATH-INFO* based on our specified rules. Similarly, our rules specify DominoBlocker to parse other fields of the API call (i.e., *wsgi.input*, *METHOD* and *PATH-INFO*), match them against our operation typing patterns, and determine that this request attaches a port with the ID *f91398* to a VM with the ID *131*.

The aforementioned steps enable DominoBlocker to intercept management API calls and enforce the monitoring policies (Step 1 above) and collect the necessary information for provenance construction and threat prevention (Step 2 above).

Runtime provenance construction. As Fig. 13 shows, during runtime, *API Requests/Responses*

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

(a) Integrating DominoBlocker as a middleware with the networking service of OpenStack (Neutron).

```
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) Information extracted from an API call attaching a port to a VM.

Figure 14: The integration of DominoBlocker with OpenStack and the intercepted information.

Interceptor intercepts the parameters of management API calls and passes them to *Requests Processor*, which obtains the affected resources and the type of the requested operations based on our specified rules. We note that the management API calls that are sent to OpenStack services for creating a resource (e.g., the API call requesting *CreateVM* operation) do not include the ID of the created resource. Therefore, DominoBlocker also intercepts the response of such API calls that are sent from services back to management API interfaces to obtain the ID of the newly created resource. Then, *Provenance Builder* updates the provenance database implemented in Neo4j[1] based on the extracted information. We use *py2neo*[2] as an interface between our middleware and the graph query language, Cypher, for interacting with the database.

Offline forensic analysis. To facilitate identifying the root cause of an incident, the analyst initializes his/her selected pruning mechanisms (detailed in Section 3.3.3) using our graphical interface, through which the analyst selects and initiates the pruning scripts with the parameters reported about the alert (e.g., time of the detection and the ID of the target VM). The analyst may also specify a time interval so that nodes and edges generated beyond that time interval are discarded in the

[1]Neo4j Graph Database, `https://neo4j.com`
[2]`https://py2neo.org/v4`

investigation. Moreover, as Fig. 13 shows, pruning may be conducted based on security assumptions (e.g., *Network connectivity*, *Tenant-based*, etc.) about the cloud or the detected incident. For instance, DominoBlocker can be configured to prune all nodes corresponding to resources that are not in a network connectivity with the target resource or are created/updated by different tenants.

Runtime threat prevention. To conduct threat prevention, *Pattern Verifier* issues a query to the provenance graph database to detect the suspicious graph pattern specified by the monitoring policy. Once the specified graph pattern is detected in the provenance graph, the *Decision Enforcer* blocks or notifies the requested operation according to the specified enforcement action: if the enforcement action is *block*, Decision Enforcer informs our API interceptors to generate "*403 Forbidden*" response without passing the management API call to the cloud services. Moreover, Decision Enforcer removes the nodes and edges representing the newly blocked operation and its affected resources from the provenance graph. If the enforcement action is *notify*, Decision Enforcer informs the API interceptors to log a warning message and pass the management API call to the endpoint cloud services.

3.6 Evaluation

To evaluate DominoBlocker, we seek to answer the following questions:

RQ1: How effectively can DominoBlocker capture and prevent real-world attacks?

RQ2: How much does DominoBlocker affect the execution time of management operations? How does the additional delay compare to the existing techniques and vary in different cloud setups?

RQ3: How much can our pruning schemes reduce the size of the provenance graph? How does it vary based on different pruning criteria provided by the cloud admin?

RQ4: What is the resource overhead of DominoBlocker in terms of storage and CPU?

RQ5: How completely and soundly can we capture the management API calls and prune the provenance graph?

Testbed experimental setup and dataset. We run DominoBlocker on an Ubuntu 18.04 server with eight virtual CPUs and 12 GB of virtual RAM. To generate diverse datasets, we create 55 subnets connected to different numbers of VMs in each dataset, and randomly vary the operations affecting virtual resources (e.g., starting/locking VMs and creating/updating virtual ports). Table 2 shows statistics about the datasets generated in our cloud testbed. In Section 3.6.2.2, we describe our dataset collected from a real cloud environment.

Table 2: Statistics of datasets generated in our cloud testbed.

Dataset	**D_1**	**D_2**	**D_3**	**D_4**
# of API calls	2435	4885	7352	9835
# of nodes	4657	9072	13482	17904
# of edges	3513	6866	10164	13438
Total # of VMs	352	611	875	1145
Average # of VMs per subnet	5	8	11	13

Table 3: Attack scenarios used to evaluate the effectiveness of DominoBlocker. The first and second cases are showcased in Motivating Example, and the sixth case is showcased in Example 2.

	Root Cause	**Vulnerability**	**Detected Incident**	**Type of Most Relevant Management Operations**
1	Race Condition to Bypass Anti-spoofing Rules [107]	CVE-2015-5240	Data Leakage	CreatePort, CreateVM, UpdatePort
2	Proper Authorization Failure [119]	CVE-2014-0056	Data Leakage	CreateRouter, CreatePort, CreateVM
3	Failing to Update Security Groups [66]	CVE-2015-7713	Port Scanning	AddSecurityGroup, StartVM, DeleteSecurityGroupRule
4	Failing to Delete Resized VMs	CVE-2016-7498	Disk DoS	CreateVM, ResizeVM, DeleteVM
5	Soft-rebooting Migrated VMs	CVE-2020-17376	Data Corruption	CreateVM, LiveMigrateVM, SoftRebootVM, CreateVM
6	Network Misconfiguration	No CVE	Data Leakage	CreateVM, AttachSubetToRouter, AttachSubetToRouter
7	Malformed Security Group Rules	CVE-2019-9735	Host Unavailability	CreateVM, CreateSecurityGroup, AddSecurityGroupRule

3.6.1 Effectiveness

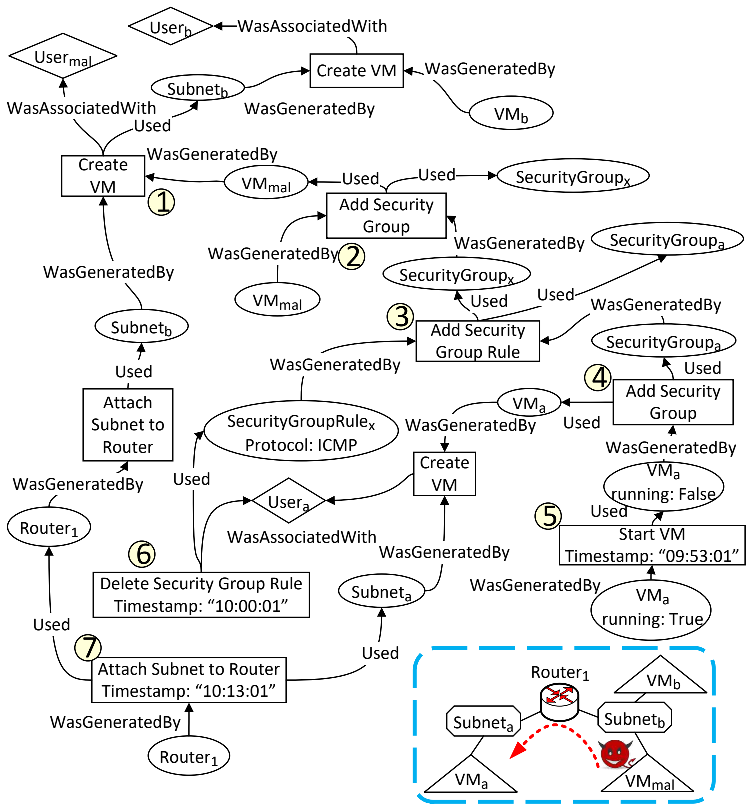
To answer RQ1, in our testbed, we automatically reproduce the attack scenario of seven security incidents that involve cloud management operations via Bash scripts[1]. Some of these attacks are discussed in existing works [66, 119, 107] focusing on cloud security verification. Table 3 summarizes those attack scenarios, the most relevant operations, and the exploited vulnerabilities. For all scenarios, DominoBlocker could successfully trace back the incidents to the root cause and prevent the recurrence of attacks. We showcase the effectiveness of our approach based on all cases: the scenario exploiting two vulnerabilities (first and second rows of Table 3) is presented in our Motivating Example; Example 2 showcases the prevention of the sixth case, and the other four cases are detailed below.

3.6.1.1 Failing to Update Security Groups

In this scenario (Row three, Table 3), the analyst receives a port scanning alert due to ICMP traffic from VM_a, which belongs to $User_a$. Fig. 15(a) shows an excerpt of the provenance graph generated by DominoBlocker. Using the provenance graph, the analyst can see that $User_{mal}$ creates VM_{mal} in $Subnet_b$ (Step 1), and adds VM_{mal} to $SecurityGroup_x$ (Step 2). The analyst can also observe that a rule ($SecuirtyGroupRule_x$) is added to $SecurityGroup_x$ to allow ICMP traffic from the VMs of $SecurityGroup_x$ (e.g., VM_{mal}) to the VMs of $SecurityGroup_a$ (Step 3). Next, $User_a$ adds VM_a to $SecurityGroup_a$ (Step 4) and starts this VM (Step 5). $User_a$ deletes $SecuirtyGroupRule_x$ (Step 6) before connecting $Subnet_a$ and $Subnet_b$ (Step 7). As deleting $SecuirtyGroupRule_x$ restricts ICMP traffic to VM_a, the port scanning incident is not supposed to happen. Hence, the analyst suspects that the deletion of $SecuirtyGroupRule_x$ was not successfully applied due to a potentially existing vulnerability (*CVE-2015-7713*[2]), which disables the update of a security group for a running VM. To prevent the recurrence of such attacks in the future, the analyst leverages the threat prevention mechanism of DominoBlocker, which allows specifying the monitoring policy shown in Fig. 15(b).

[1] `https://www.gnu.org/software/bash/manual/bash.html`
[2] `https://nvd.nist.gov/vuln/detail/CVE-2015-7713`

During runtime, DominoBlocker verifies the provenance graph against the specified graph pattern and notifies the admin once a user attempts to delete a security group rule applied to a running VM. This enables the admin to closely verify that the deletion of the rule is applied prior to allowing subsequent network connections made to such VMs.



(a) Root cause of the port scanning incident identified using DominoBlocker.

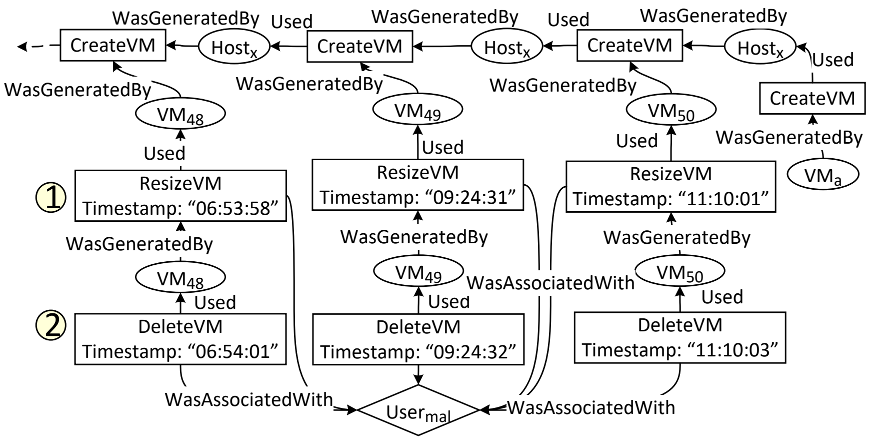
```
Pattern: {
MATCH path = (a{EventType:'DeleteSecurityGroup'})-[:Used]->(b{ResourceType:'VM',
uuid:$VM-ID})-[*]->(c{ResourceType:'VM', uuid:$VM-ID})-[:WasGeneratedBy]-
>(d{EventType:'StartVM'})
WHERE ANY (node_on_path IN NODES(path) WHERE node_on_path.type <>
'StopVM')
RETURN a}
action: notify
```

(b) An example of a monitoring policy to detect and notify the admin once a security group rule applied to a running VM is updated.

Figure 15: Applying our solution to investigate and prevent the port scanning incident (Row three, Table 3).

3.6.1.2 Failing to Delete Resized VMs

In this scenario (Row four, Table 3), the analyst receives a disk denial of service alert once users attempt to create VMs on $Host_x$. Using the provenance graph generated by DominoBlocker (as shown in Fig. 16(a)), the analyst can observe that a user (i.e., $User_{mal}$) resizes several VMs (Step 1), and triggers another request for deleting those VMs after a few seconds (Step 2). Since deleting a VM right after it has been resized is not a routine behavior in that cloud environment, while it is conducted repeatedly by $User_{mal}$, the analyst suspects that $User_{mal}$ exploits a potential vulnerability (*CVE-2016-7498*[1]) to keep $Host_x$ occupied by preventing the actual deletion of resized VMs. Based on this observation, the analyst specifies a monitoring policy that blocks the requests for deleting VMs immediately after they are resized (Fig. 16(b)).



WasGeneratedBy
Used
CreateVM
Hostx
WasGeneratedBy
VM48
VM49
VM50
VMa
ResizeVM Timestamp: "06:53:58"
ResizeVM Timestamp: "09:24:31"
ResizeVM Timestamp: "11:10:01"
DeleteVM Timestamp: "06:54:01"
DeleteVM Timestamp: "09:24:32"
DeleteVM Timestamp: "11:10:03"
1
2
WasAssociatedWith
Usermal

(a) Root cause of the disk denial of service incident identified using DominoBlocker (some edges and nodes are omitted for the sake of readability).

```
Pattern: {
//to detect the deletion of a resized VM
MATCH (op1:'ResizeVM')<-[]-(resouce1)<-[*]-(resource2)<-[]-(op2:'DeleteVM'})
WHERE op2.TimeInSecond-op1.TimeInSecond < $DeltaTimeInterval
AND resource1.id = resource2.id
RETURN op1, resource1, op2}
Action: block
```

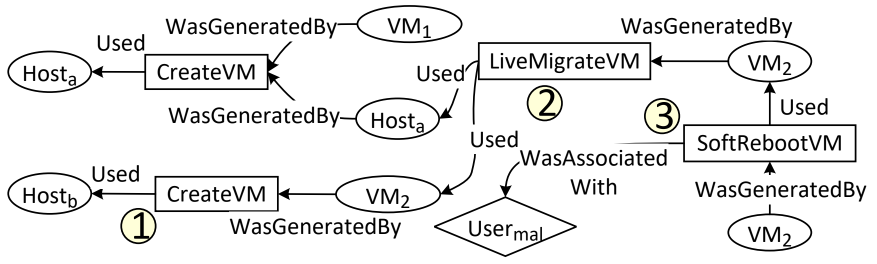
(b) An example of a monitoring policy to detect and block the deletion of a VM immediately after it is resized.

Figure 16: Applying DominoBlocker to investigate and prevent the disk denial of service incident (Row four, Table 3).

[1]`https://nvd.nist.gov/vuln/detail/CVE-2016-7498`

3.6.1.3 Soft-rebooting Migrated VMs

In this scenario (Row five, Table 3), the analyst initiates an investigation to identify the root cause of the corruption of VM_1. The analyst cannot explain the incident by investigating the configurations of VM_1: he/she realizes that VM_1 is not in network connectivity with other VMs, and only the admin user is privileged to access it. Therefore, VM_1 is not corrupted by other VMs sharing the same network or users logged into this VM. On the other hand, after investigating the provenance graph generated by DominoBlocker (Fig. 17(a)), the analyst realizes that VM_1 is hosted by $Host_a$ which also hosts VM_2. The analyst can also see that VM_2 is created on a different host, i.e., $Host_b$ (Step 1), and later, is migrated to $Host_a$ (Step 2). Next, another user (i.e., $User_{mal}$) soft-reboots VM_2 (Step 3) to exploit a vulnerability (*CVE-2020-17376*[1]), which attaches the storage volume of VM_1 to VM_2 on $Host_a$. Therefore, $User_{mal}$ is allowed to corrupt the data of VM_1, as it is accessible through VM_2. To prevent such incidents in the future, the analyst specifies the monitoring policy shown in Fig. 17(b), which notifies the admin once a user attempts to soft-reboot a migrated VM.



(a) Root cause of the data corruption incident identified using DominoBlocker.

```
Pattern: {
//to detect the soft-reboot of a live-migrated VM
MATCH (op1:'LiveMigrateVM')<-[]-(resouce1)<-[*]-(resource2)<-[]-(op2:'SoftRebootVM'})
AND resource1.id = resource2.id
RETURN op1, op2}
Action: notify
```

(b) An example of a monitoring policy to detect and notify the admin once a live-migrated VM is rebooted (soft).

Figure 17: Applying DominoBlocker to investigate and notify the data corruption in a VM (Row five, Table 3).

[1] `https://nvd.nist.gov/vuln/detail/CVE-2020-17376`

3.6.1.4 Malformed Security Group Rules

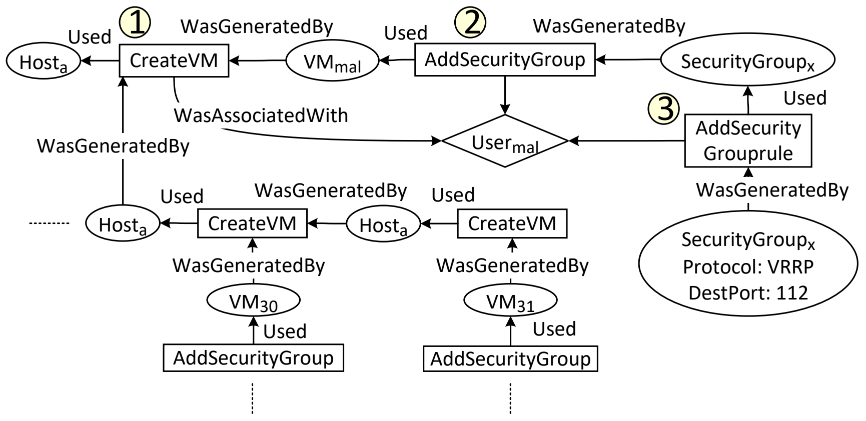
In this scenario (Row seven, Table 3), the analyst receives a performance alert from most VMs located on $Host_a$. Fig. 18(a) shows an excerpt of the provenance graph generated by DominoBlocker. Using the provenance graph, the analyst can see that a user, i.e., $User_{mal}$, creates a VM on $Host_a$ (Step 1), and attaches this VM to $SecurityGroup_x$ (Step 2). Next, $User_{mal}$ specifies a malformed security group rule allowing *VRRP* network traffic to the destination port 112 (Step 3). As specifying the destination port number for VRRP network traffic is not supported by OpenStack [5], the analyst suspects that $User_{mal}$ specifies such a rule to exploit a vulnerability (*CVE-2019-9735*[1]), which disables the security group rules of all VMs on $Host_a$, and leaves them prone to malicious resource-intensive network traffic. Therefore, the analyst specifies a monitoring policy (Fig. 18(b)) which notifies the cloud admin once a user indicates the destination port number for the specified security group rules.

Based on our evaluated attack scenarios, we can conclude that DominoBlocker is effective in assisting the analyst to identify the root cause of security incidents and automatically preventing those incidents in the future.

3.6.2 Efficiency

To answer RQ2, we measure the delay caused by DominoBlocker in the runtime execution of cloud operations (i.e., the elapsed time between sending a request from management interfaces and the completion of its execution). The additional runtime delay is incurred by our provenance construction and threat prevention modules. The time required for the communication between DominoBlocker and Neo4j database server is also considered in our measurements. We also compare the delay introduced by DominoBlocker with the delay incurred by two other approaches: the execution time of cloud operations (with no security solution applied) and the delay introduced

[1]`https://nvd.nist.gov/vuln/detail/CVE-2019-9735`



(a) Root cause of the host unavailability incident identified using DominoBlocker.

```
Pattern: {
//to detect the addition of a security group rule with a specified destination address
MATCH (op1:'AddSecurityGroup')<-[*]-(op2:'AddSecurityGroupRule')<-[]-(r:'SecurityGroupRule')
WHERE EXISTS(r.DesPort)
RETURN op2}
Action: notify
```

(b) An example of a monitoring policy to detect and notify the admin once a destination port is specified for an added security group rule.

Figure 18: Applying DominoBlocker to investigate and notify the host unavailability (Row seven, Table 3).

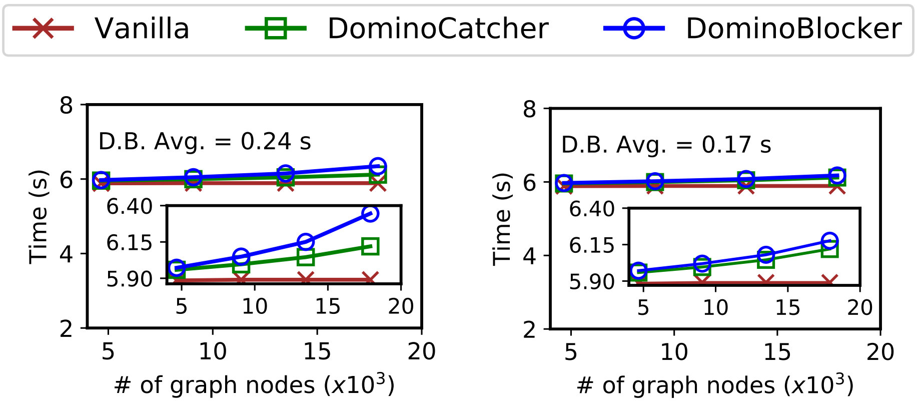
by a more basic version of our work, DominoCatcher [106], which does not support threat prevention. To evaluate both regular and label-based threat prevention mechanisms, we conduct our evaluations for our Motivating Example and Example 2, which leverage regular (Fig. 8) and label-based (Fig. 12(a)) monitoring policies, respectively. We conduct our experiments based on both our testbed OpenStack cloud and a real cloud.

3.6.2.1 Experiments with Cloud Testbed

In this experiment, we evaluate the effect of the size and virtual network topology of the cloud on the performance of DominoBlocker.

Effect of cloud size. To evaluate the efficiency for clouds having different sizes, we simulate attack scenarios targeting VMs attached to random subnets of each dataset. Figures 19(a) and 19(b) compare the runtime delay experienced by users without applying a security solution (Vanilla), after applying DominoCatcher [106] and DominoBlocker to prevent the attack scenarios of our Motivating Example (Section 3.1) and Example 2 (Section 3.4.2). DominoCatcher adds an average delay of around 0.14 seconds to the execution time of operations for constructing the provenance graph (without supporting threat prevention). DominoBlocker introduces the average accumulative delays of 0.24 and 0.17 seconds for provenance construction and threat prevention based on the monitoring policies of our Motivating Example and Example 2, respectively. The delay grows with the size of the provenance graph, as locating nodes (for creating new dependencies in provenance construction) and graph patterns (for threat prevention) are more time-consuming in larger provenance graphs. Furthermore, our label-based threat prevention approach causes smaller delays (0.17 seconds in average) than our regular threat prevention approach, since the former only compares the monitoring labels of the newly connected nodes (detailed in Section 3.4.2.3). In contrast, conducting regular threat prevention requires tracing back long paths of the provenance graph, which subsequently incurs a longer delay (around 0.24 seconds in average).

We also measure the ratio between the additional delay caused by DominoBlocker and the average execution time of management operations. Table 4 shows that in all datasets, the incurred additional overhead is less than 8% of the execution time of management operations, which demonstrates the scalability of our approach.



(a) Data leakage prevention (Motivating Example).

(b) Network isolation (Example 2).

Figure 19: Comparing the delay of DominoBlocker with Vanilla and DominoCatcher [106] approaches based on our (a) Motivating Example and (b) Example 2 (*D.B. Avg.* stands for the average additional delay experienced by users applying DominoBlocker).

Table 4: The runtime overhead imposed by DominoBlocker in different size of clouds with respect to the execution time of management operations.

		Threat Prevention	
Dataset	**Provenance Const.**	**Data Leakage**	**Network Isolation**
D_1	1.25%	1.49%	1.47%
D_2	1.81%	2.69%	2.20%
D_3	2.64%	4.42%	3.23%
D_4	3.90%	7.74%	4.87%

Different management operations. Fig. 20 compares the average delay caused by DominoBlocker with the execution time of different management operations (Vanilla) in all datasets (shown in Table 2). We observe that the delay incurred by DominoBlocker (maximum 8.07% average overhead) is negligible compared with the execution time of management operations. Table 5 shows the total additional delay incurred by DominoBlocker in datasets of different sizes. The maximum delay is around 0.24 seconds for capturing the operations in the provenance graph and verifying the specified monitoring policy during runtime. Operations affecting multiple resources have a longer delay, as adding such operations to the provenance graph requires locating and creating multiple resource nodes and dependencies, which is more time-consuming. For instance, operation *Attach-Port-To-VM* (encoded as *Operation 4* in Table 5) has the highest overhead, since for adding this operation to the provenance graph, DominoBlocker creates two edges between *Attach-Port-To-VM* operation node and the nodes representing the affected VM and port. In summary, DominoBlocker incurs a negligible additional overhead for all cloud management operations.

Table 5: The additional delay (in seconds) incurred to different management operations (numbered as specified in Fig. 20).

Dataset/Operation	**1**	**2**	**3**	**4**	**5**	**6**
D_1	0.087	0.086	0.087	0.088	0.09	0.09
D_2	0.15	0.158	0.156	0.163	0.163	0.161
D_3	0.262	0.260	0.254	0.264	0.263	0.259
D_4	0.458	0.448	0.459	0.462	0.453	0.454
Average	0.24	0.238	0.239	0.244	0.242	0.241

Effect of virtual network topology in the cloud. To evaluate the effect of network topology, in

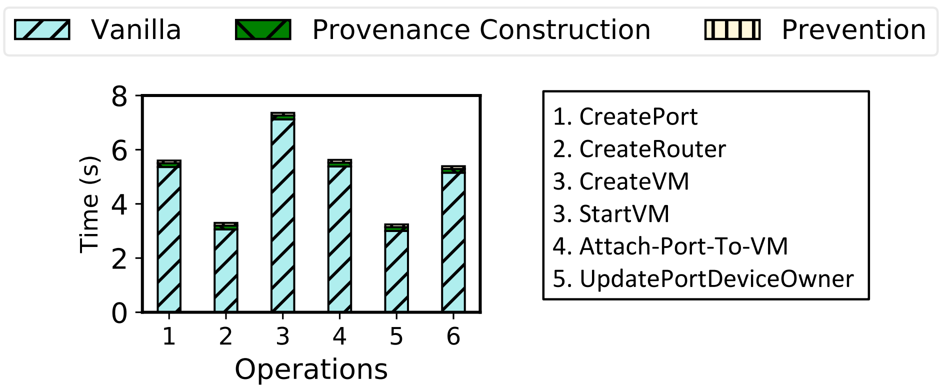


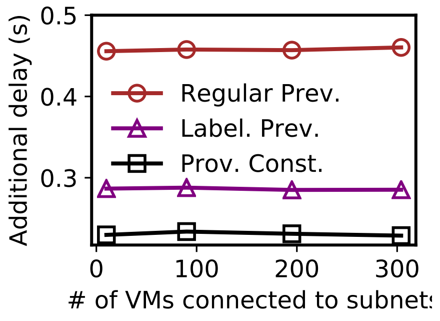
Figure 20: Evaluating the delay incurred to different operations.

this work, we simulate attack scenarios targeting VMs attached to subnets of four different sizes in a provenance graph with 19,254 nodes. Table 6 shows the number of VMs and operations affecting resources (e.g., VMs and ports) attached to each subnet. Fig. 21(a) shows that, in all subnets, the delay of the provenance construction is around 0.229 seconds and the delay caused by the threat prevention mechanism is around 0.455 and 0.286 seconds, for Motivating Example (regular prevention) and Example 2 (label-based prevention), respectively. The delay does not vary drastically for different subnets, since the complexity of paths connected to all subnets (e.g., the number of edges connected to operation nodes) is similar due to the small difference between the number of resources affected by different operations. Nevertheless, without leveraging monitoring labels (i.e., regular prevention in Fig. 21(a)), the delay slightly increases with the size of subnets (from 455 milliseconds for the smallest subnet to 460 milliseconds for the largest one), while our label-based prevention mechanism has a similar delay in all subnets. One reason is that, based on the regular monitoring policies, our threat prevention module traces back graph paths which are longer for subnets with a larger number of resources in our datasets. In contrast, the additional delay introduced by label-based threat prevention is similar in all subnets, since this approach verifies the label of the last node on a path (instead of tracing back long paths), which shows the benefit of our label-based prevention mechanism.

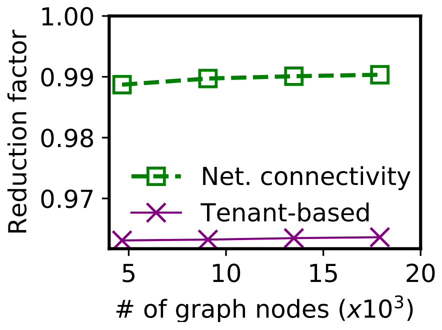
We conclude that DominoBlocker incurs a negligible runtime overhead to prevent attacks targeting clouds with different sizes or virtual network topology.

Table 6: The statistics of subnets used for evaluating the effect of cloud topology and dynamicity.

	Subnet			
# of connected VMs	10	90	195	304
# of management operations	30	617	1349	1971



(a) Effect of network topology.



(b) Size reduction.

Figure 21: Evaluating the (a) overhead in clouds with different network topology and (b) size reduction of DominoBlocker.

3.6.2.2 Experiments with Real Cloud

To evaluate the applicability of DominoBlocker in real cloud environments, we measure its efficiency using the data collected from a research cloud hosted at one of the largest telecommunications vendors with hundreds of hosts and users. We note that since the installation of our data collection mechanism was not allowed in this cloud, we construct the provenance graph based on the logs automatically generated by the cloud infrastructure (although such logs lack sufficient information for provenance construction [67]). Our collected logs correspond to 1,148 management API calls in the cloud environment of 354 VMs. Our constructed provenance graph consists of 2,157 nodes and 1,565 edges. On average, the delay incurred by DominoBlocker is around 0.037 seconds to add nodes and edges to this provenance graph representing the newly requested operations and affected resources. DominoBlocker causes the average delay of 8.4 and 8.1 milliseconds to conduct threat prevention based on our regular (Motivating Example) and label-based (Example 2) monitoring policies, respectively. We note that this delay is smaller than the delay measured in our testbed dataset as this provenance graph is smaller than the provenance graphs constructed based on our testbed cloud.

3.6.3 Size Reduction of Provenance Graph

To answer RQ3, we evaluate the effectiveness of our pruning schemes in reducing the size of the provenance graphs based on two pruning criteria shown in Fig. 13: network connectivity (discarding resources that are not in direct network connectivity with the target resource) and tenant-based (discarding resources affected by different cloud tenants). Fig. 21(b) depicts the average reduction for datasets of different sizes, where the reduction factor is the number of pruned nodes over the total number of nodes. We can see that pruning based on network connectivity reduces the size of the provenance graph by almost 99% in all datasets. The reason for this significant reduction is that the number of resources in network connectivity with the target resource is much less than the total number of resources. Therefore, pruning nodes of resources connected to other networks significantly reduces the size of the provenance graph. Likewise, pruning nodes corresponding to different tenants drastically decreases the size of the provenance graph (to around 4% of the original size), since the number of nodes representing the operations and resources that are issued or affected by each tenant is significantly smaller than the total number of graph nodes. We note that although the measured reduction factor may vary for clouds with a different number of connected resources or tenants, our results provide a useful approximation of the effectiveness of our pruning scheme.

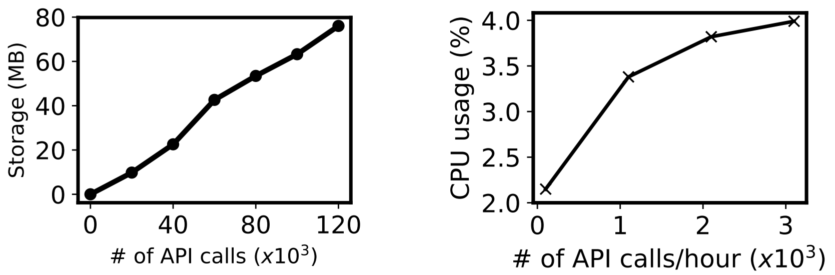
In summary, our pruning scheme effectively assists the analyst under various potential criteria by remarkably narrowing down the root cause investigation.

3.6.4 Resource Overhead

To answer RQ4, we measure the storage and CPU overhead caused by DominoBlocker.

Storage consumption. Fig. 22(a) shows that for the provenance graph constructed by 120,000 management API calls, only 80-megabyte storage is required. This number of management API calls is much higher than the number of management API calls issued in one day in a real cloud reported in [119], which indicates that the storage cost of DominoBlocker is acceptable.

CPU consumption. We also evaluate the CPU overhead caused by DominoBlocker at different rates of intercepted management API calls. In our experiments, we vary the rates of API calls such that the time interval between each pair of consecutive API calls is a fraction of the time interval between the same management API calls in the logs collected from our real research cloud platform. Fig. 22(b) shows that the average CPU usage of DominoBlocker increases almost linearly with the rate of intercepted management API calls. The rate of 500 API calls per hour is comparable to the rate of API calls issued in our real research cloud, which incurs less than 2.5% CPU overhead. Moreover, DominoBlocker causes less than 4% CPU consumption while intercepting around 3,000 API calls per hour, which demonstrates the scalability of our solution in clouds with higher rates of configuration changes.



(a) Storage cost. (b) CPU consumption.

Figure 22: Evaluating the resource overhead of DominoBlocker.

3.6.5 Correctness

To answer RQ5, we evaluate how complete and sound DominoBlocker is in terms of capturing changes in cloud environments and discarding irrelevant operations.

Completeness. All operations directed through cloud management interfaces are passed as API calls to the endpoint services (e.g., computing, networking, image, and storage services). Hence, by deploying DominoBlocker as middlewares attached to all managerial cloud services, we enable 100% coverage by capturing all management API calls. Moreover, processing the request body of the intercepted API calls allows identifying all resources affected by each operation. We show the number of unique types of API calls issued to the most commonly used cloud services (as specified by OpenStack API documentation [3]) in Table 7.

Table 7: The coverage of DominoBlocker for the unique management API calls issued to different cloud services.

Cloud service	Compute	Network	Object storage	Image
# of unique API calls	313	251	16	31
Our coverage (%)	100	100	100	100

Soundness. To ensure the soundness of provenance analysis while following graph paths, DominoBlocker stores the interception time of management API calls as a property in operation nodes. Moreover, all edges except *WasAssociatedWith* point to the past, which captures the temporal order between nodes in a path. To preserve the soundness of our pruning schemes, DominoBlocker prunes only the nodes and edges that are identified to be irrelevant to the detected incident based on the pruning criteria provided by the cloud admin. Furthermore, DominoBlocker follows paths in both forward and backward directions to identify the nodes that are potentially involved in preparing the condition for the incident or are affected by the attack, and prune other nodes.

3.7 Discussion

In this section, we discuss the scope and future directions of DominoBlocker.

Applicability to other platforms. Although this work focuses on the OpenStack cloud, DominoBlocker can be applied to other cloud platforms with an initial effort of adapting to their management operations and virtual resources. Furthermore, we have studied the applicability of our data collection and enforcement mechanisms in different clouds. For example, in the Amazon cloud, it is possible to intercept and validate API calls before they are received by the endpoint application using AWS Lambda plug-in [67]. Our management-level provenance model can also be applied to Kubernetes (a major container orchestrator) [89] to capture the causal relationship between management operations.

Integrating with other provenance-based solutions. DominoBlocker can be integrated with OS-level provenance solutions to narrow down analysis on low-level system calls. For example, the analyst may leverage DominoBlocker to identify the exploited resources, and subsequently, apply OS-level provenance solutions to analyse the events affecting such resources. Moreover, we will leverage existing provenance-based techniques [118, 42] to automatically detect the chain of potentially malicious operations through identifying anomalous paths of the provenance graph. We also plan to investigate the effect of mimicry attacks complicating provenance analysis and apply adversarial machine learning techniques for studying such attacks.

Distributed Systems. DominoBlocker can support provenance analysis in distributed systems (e.g., several controller nodes) by deploying our middlewares on each node capturing and communicating provenance metadata with a central node and database. However, similar to most existing provenance solutions [92, 81], the support for distributed systems is coupled with some challenges (e.g., the runtime delay caused by transmitting provenance records, the need for secure transmission protocols, etc.). In our future work, we will evaluate the effect of such challenges on the performance of DominoBlocker.

3.8 Related Work

Provenance-based security solutions have been explored by many existing works (e.g., [49, 92, 40, 91, 42, 117, 112]). Most of these solutions focus on capturing system transformations through tracing low-level system calls. For instance, King et al. [49] leverage OS-level provenance graphs for investigating security incidents in operating systems. Hi-Fi [94] proposes a kernel-level provenance-based solution to monitor malicious behavior. LPM [16] applies provenance analysis to ensure the authenticity of communications. CamQuery [92] enables efficient runtime provenance analysis (e.g., data loss prevention) by tracing userspace and in-kernel executions. Network provenance-based solutions (e.g., [23, 22]) focus on reference packet events and network traffic. The authors in [121] leverage *negative provenance* to explain the absence of events. Multi-layer provenance solutions (e.g., [62, 74, 44]) enable more accurate analysis by integrating application logs into the OS-level provenance graphs. Nodoze [43] and ProvDetector [118] focus on triaging alerts and detecting malicious programs, respectively, in enterprise environments. The authors in [13] propose a solution to build a model of attack signatures using sequence learning. Although most of those solutions can be extended to clouds, in contrast to our work, they cannot directly enable identifying the management operations leading to security incidents. There exist solutions (e.g., [124, 72]) focusing on summarizing OS-level provenance graphs at a higher abstraction level based on human expertise and machine learning techniques. However, unlike our work, those solutions cannot directly trace cloud management operations, and DominoBlocker can complement those solutions by tracing cloud configuration changes at a higher abstraction level (i.e., management operations) without relying on error-prone human expertise for such abstraction.

There also exist provenance-based threat detection and prevention solutions applied to virtualized environments. In [40], the authors propose an anomaly detection solution based on low-level system calls for containers. CLARION [25] is a solution for generating precise provenance graphs tracing system calls across different namespaces in container environments. Winnower [42] enables more scalable threat detection by inducing the Deterministic Finite Automata (DFA) representing the provenance graphs of replicated container applications, and prevents the recurrence of the detected threats. In [117], the authors propose a provenance-based solution for investigating and preventing incidents in the Internet of Things (IoT) environments. The authors in [115] leverage data provenance to conduct flow-level forensics in software defined networking (SDN) environments. ProvSDN [112] is a provenance-based solutions that prevents the flow of sensitive information to unprivileged applications. Lu et al. [59] leverage data provenance to investigate the data access, and Bates et al. [15] propose a provenance-based access control approach for ensuring storage security in clouds. In [81], the authors propose a tenant-aware solution for enhancing access control in OpenStack. DominoBlocker can work in tandem with those solutions (as explained in Section 3.7) by tracing the information flows between virtual resources at a higher abstraction level.

3.9 Conclusion

Preventing security incidents is crucial for ensuring the secure adoption of cloud computing. In this work, we developed DominoBlocker, the first management-level provenance solution for preventing security incidents in clouds. DominoBlocker leveraged the concept of data provenance to find the management operations leading to attacks in cloud virtual infrastructures and provided efficient pruning mechanisms to pinpoint the root causes and prevent future recurrent incidents. We integrated DominoBlocker to OpenStack and demonstrated the effectiveness of our approach based on real-world attack scenarios. Moreover, our evaluations using our diverse testbed datasets showed that DominoBlocker introduces a negligible overhead. As future work, we plan to integrate our framework with low-level provenance-based techniques. Furthermore, we will use machine learning techniques for detecting potentially malicious operations during runtime.

Chapter 4

ProvTalk: Towards Interpretable Multi-level Provenance Analysis in Networking Functions Virtualization (NFV)

4.1 Introduction

Today, softwarized services are increasingly deployed over virtual resources (e.g., containers or VMs) sharing underlying physical infrastructures [114, 83]. In particular, NFV enables the replacement of proprietary devices with software network services and allows for more dynamic and agile network service deployment in the cloud [31]. This new paradigm converts traditional networking into a multi-level NFV stack by running virtual services over multiple levels of virtual resources. Each level of the NFV stack is operated by a different managerial component accessible through its API interface to create, modify or delete network services and their related resources. Such added complexity of NFV may increase the risk of vulnerabilities and misconfiguration in the deployed services [76, 96]. For instance, by exploiting `CVE-2020-12689` [45], an attacker can gain unauthorised access to the NFV management API, and compromise other clients' network services. The multi-level nature of NFV together with its sheer scale and complexity may also render pinpointing the root cause of security incidents more challenging. Existing solutions in NFV (e.g., [100, 53]) mainly focus on localizing failed components instead of identifying the activities leading to the incident.

Unique challenges of provenance analysis in NFV. Data provenance is a well-established technique used for root cause analysis that has been applied in other domains such as IoT (e.g., [117]), SDN (e.g., [115]), cloud (e.g., [106]) and operating systems (e.g., [118]). Most existing approaches rely on tracking system-level events (detailed in Section 4.9.3 and 4.10). The existing management-level solution [106] is limited to clouds with no support for multi-level/cross-level analysis (more comparison in Section 4.9.3.3). However, applying provenance analysis to each level of the NFV stack (e.g., cloud), while ignoring the relationships between levels, would be insufficient. For instance, a user request for creating a network service may lead to a series of operations to create virtual resources across different levels of the NFV stack. Without capturing the dependencies between such operations and resources, a provenance analysis would not be able to link a security incident to its root cause if they happen to be at different levels of the NFV stack.

There exist provenance analysis solutions for other multi-level systems (e.g., SDN). However, one unique aspect of NFV that distinguishes it from those systems is that different NFV levels are actually representing the same components (e.g., a virtual firewall) with different degrees of abstraction (e.g., as a service, a virtual network function, or a virtual machine) as detailed in Section 2.3. In contrast, most existing multi-level provenance solutions (e.g., [62, 74, 44]) mainly focus on multiple systems working together while each plays a different role (e.g., SDN controller vs. applications [112], or the operating system of the host vs. applications [44]). Therefore, their provenance graphs do not need to be explicitly segregated into different levels that can be mapped back to each other. In other words, although those solutions can analyse the interactions between different systems, they do not support the need for analysing the information flows to/from different *abstractions* of the *same* resources that we face in NFV (which will be further illustrated through a concrete example in Section 4.2.2).

Moreover, the sheer scale of NFV environments implies impractically large and complex provenance graphs for human analysts to interpret. Some recent works (e.g., [124, 72]) focus on assisting the analyst by identifying the high-level abstraction of behavior corresponding to different parts of the provenance graph. However, these approaches still require some level of domain-knowledge. For instance, the analyst may be required to determine a label (e.g., program compilation) for each extracted subgraph in the training dataset [124]. Determining such labels can be especially challenging in a multi-tenant environment like NFV considering the complexity and interleaving nature of the users' behavior.

Key ideas. In this work, we argue that the uniqueness of NFV not only leads to novel challenges in provenance analysis but also brings about new opportunities. To explore such opportunities, we propose ProvTalk [109], an interpretable multi-level provenance analysis approach for assisting security analysts to investigate the root cause of security incidents in NFV. The key insight behind ProvTalk is that the multi-level aspect of NFV intrinsically provides higher level semantics corresponding to lower-level concepts (e.g., a virtual firewall is represented as a virtual network function or virtual machine at lower levels). By establishing such a cross-level mapping, we can trace a security incident back to its root causes located at a different level, while improving the interpretability of the provenance graph. Specifically, ProvTalk *links* the provenance graphs at different levels of the NFV stack through capturing the cross-level dependencies between different abstractions of the same network service, which also implicitly captures the cross-level dependencies among operations. Then, based on the captured dependencies, ProvTalk improves the interpretability of its results in three steps. First, ProvTalk removes irrelevant information from provenance graphs by propagating the pruning label of nodes across different levels based on the established dependencies between resources and operations. Second, through mining (system or user-related) frequent patterns, ProvTalk hides redundant nodes by visually aggregating them into a single node (which can be expanded to reveal the hidden details when necessary). Third, ProvTalk leverages a rule-based approach to automatically translate details of a provenance graph into a human-readable format to provide high-level guidance to analysts.

In summary, our main contributions are as follows.

- To the best of our knowledge, ProvTalk is the first provenance-based solution specifically designed for NFV. Our provenance model captures the unique multi-level nature of NFV environments, and provenance analysis using ProvTalk allows tracing a security incident back to its root cause potentially located at a different level.

- We propose three novel techniques to improve the interpretability of provenance graphs. The multi-level pruning and mining-based aggregation schemes can both reduce the size and complexity of provenance graphs, and the rule-based translation can provide useful guidance to analyzing provenance graphs. These techniques can not only ease the task of human analysts in applying ProvTalk to large scale NFV environments, but also be potentially applied to other multi-level virtual environments.

- We implement ProvTalk and integrate it into our Tacker-OpenStack [88] testbed as an attached middleware. We validate the effectiveness of ProvTalk based on real-world security incidents. Our experiments using both real-world data and testbed data show that ProvTalk produces more interpretable provenance graphs with significant reduction in their sizes, without losing the information vital for the investigation. We demonstrate that ProvTalk can capture all management API calls while incurring an insignificant storage, latency and computational overhead. Finally, our user studies show that ProvTalk can markedly ease the analysis task of real-world users.

The remainder of this chapter is organized as follows: Chapter 4.2 provides NFV background and motivates our solution. Chapter 4.3 describes our methodology. Chapter 4.8 details the implementation of ProvTalk, and Chapter 4.9 presents our experimental results and user studies. Chapter 4.10 discusses limitations and future work. Chapter 4.11 reviews related work. Chapter 4.12 concludes the report.

4.2 Threat Model and Motivating Example

This section defines our threat model and describes a motivating example for ProvTalk.

4.2.1 Threat Model and Assumptions

Our in-scope threats include both external attackers who exploit existing vulnerabilities in the NFV stack, and insiders, such as NFV clients, cloud users and tenant administrators, who make the NFV stack exploitable either through mistakes or by malicious intentions. As our provenance model focuses on capturing management operations, we limit our scope to attacks that involve operations directed through the NFV management interfaces (e.g., command line and dashboard). Similar to most existing provenance solutions, we assume our solution is deployed by the owner of the system, and thus it has access to the full NFV stack, and we assume the NFV stack management modules, the provenance building mechanism and the provenance storage are all protected with existing techniques such as remote attestation [57, 101], hash-chain-based provenance storage protection [41] or type enforcement [16].

Out-of-scope threats include attacks that involve no management operations or resources captured in the provenance graph, and attacks that can completely bypass the NFV management interfaces. Moreover, as with most works on provenance analysis, we do not consider attackers who can temper (either through attacks or by using insider privileges) the infrastructure management system (e.g., breaching the integrity of the API calls and databases of services) or the provenance solution itself. Although our framework provides more interpretable information for easier analyses, it relies on the human analysts to pinpoint the root cause at the end. Finally, although our provenance results may 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.

4.2.2 Motivating Example

In this section, we provide a motivating example to show the benefit of applying ProvTalk. Fig. 23 illustrates an attack scenario (left) and the challenges faced by a security analyst (middle and right) in investigating the root cause.

Tedious log investigation. Upon receiving an alert from the virtual IDS (VNF_{ids}) about unauthorized SSH traffic, an analyst begins investigating the root cause of this incident. As shown in Fig. 23 (middle), the analyst may need to inspect thousands of log entries at all levels of the NFV stack. However, this can be cumbersome if done manually, since there is no apparent relationship between those entries.

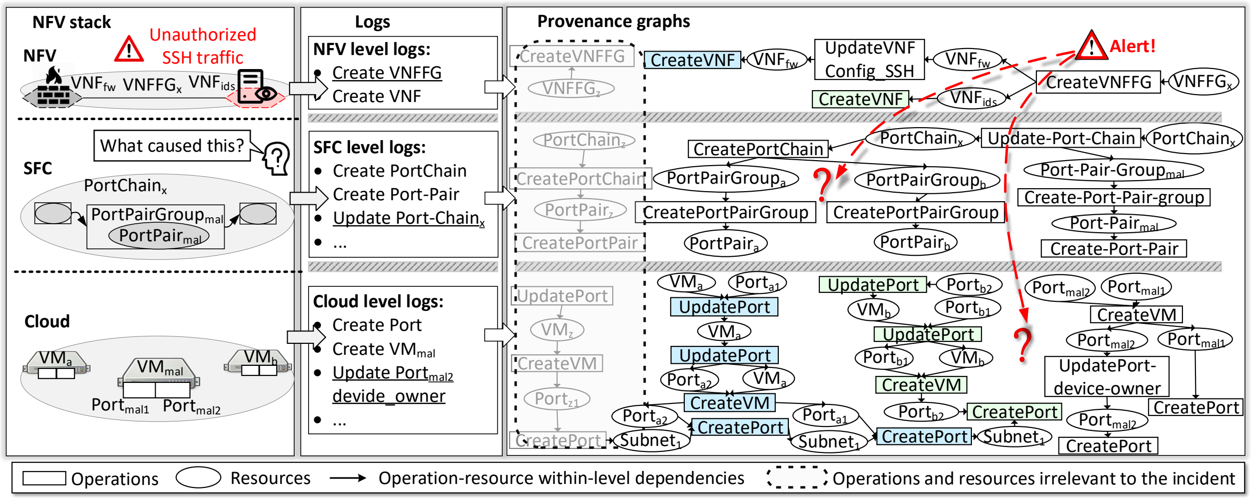


Figure 23: Attack scenario detected at the NFV-level (left); log-based analysis (middle); excerpts of the provenance graphs (right).

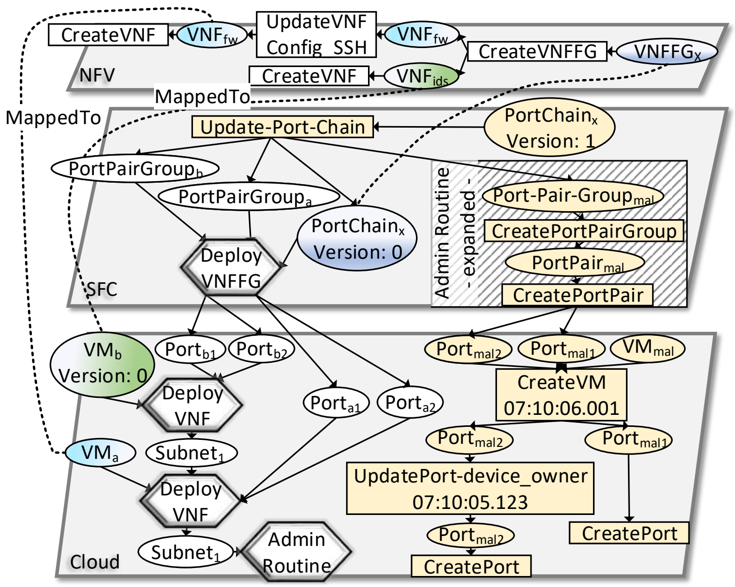
Lack of cross-level dependencies. To establish the relationships between log entries, assume the analyst applies a provenance analysis tool (e.g., [49, 106]) to each level. As shown in Fig. 23 (right), the tool would generate a provenance graph that shows the virtual IDS (VNF_{ids}) becomes part of $VNFFG_x$ through *CreateVNFFG* operation, where it is preceded by a virtual firewall (VNF_{fw}). The analyst can also see that VNF_{fw} is configured to filter the SSH traffic (by *UpdateVNF-Config-SSH* operation), and thus, the security incident (unauthorized SSH traffic) is not supposed to happen. At this point, the analyst is unable to proceed further using the provenance graph at the NFV-level alone, since no other operations at this level can explain what caused the incident.

Size and complexity of multi-level analysis. For the sake of this example, now suppose the analyst manually establishes the cross-level dependencies based on his/her experiences. For instance, he/she can identify that $VNFFG_x$ is deployed as $PortChain_x$ at a lower (SFC) level. He/she could then link the provenance graphs at different levels to each other, and continue the investigation at the lower (SFC and cloud) levels. However, the sheer scale of NFV environments and the existing dependencies among a large number of resources (e.g., there may be hundreds of VMs attached to $Subnet_1$) mean that pinpointing the root cause among all the nodes and edges in the provenance graph is still very challenging. On the other hand, the provenance graph contains a lot of irrelevant or redundant information. For instance, in Fig. 23 (right), the grayed out portion of the provenance graph is actually irrelevant to the incident, since it corresponds to an irrelevant VNFFG ($VNFFG_z$). Moreover, the groups of green and blue nodes at the cloud-level are triggered by the platform after each NFV-level operation *CreateVNF*, and thus are redundant. Leaving such information as-is in the provenance graph can make the analysis time consuming and error-prone.

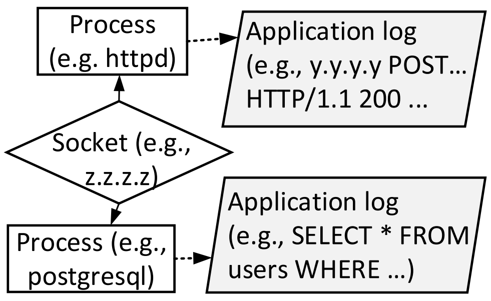
Example output of ProvTalk. ProvTalk is designed to address all the aforementioned challenges in an automatic fashion, so analysts can focus on the most relevant information for identifying the root cause. For example, Fig. 24a illustrates the result of ProvTalk corresponding to the above example. While we leave the details (e.g., the hexagons and hatched box) to section 4.3, the attack scenario is self-explanatory from the figure as follows. First, the attacker creates two ports (i.e. $Port_{mal1}$ and $Port_{mal2}$). Next, he/she updates the *device_owner* field of $Port_{mal2}$, and immediately creates VM_{mal} attached to this port so that he/she can exploit a vulnerability [86] for spoofing the IP address of the enterprise sending network traffic into $VNFFG_x$. Finally, the attacker inserts the port pair group of VM_{mal} into the port-chain corresponding to $VNFFG_x$ (via *Update-Port-Chain* operation). The attacker can then send malicious traffic using VM_{mal} inserted between the virtual firewall and IDS services to evade them.

Comparison to existing works. In Fig. 24a, we can see the provenance graph generated by

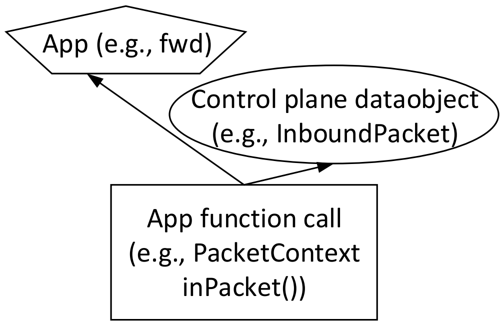
ProvTalk is explicitly divided into three disjoint levels, with nodes corresponding to a common resource "mapped to" each other (e.g., VNF_{fw} vs. VM_a). In contrast, Fig. 24b and 24c show that the provenance graphs of existing "multi-tier" provenance techniques (e.g., [44, 112]) usually do not have such explicit separation between levels, as they focus more on the information flow between multiple systems that play different roles (e.g., operating system vs. application in Fig. 24b, or application vs. control plane in Fig. 24c). Therefore, such works are not designed to support the need for capturing the information flow to/from different abstractions of the same resource in NFV.



(a) The result of ProvTalk for the attack scenario of Fig. 23 (details in Section 4.6). Nodes visualizing the same resource have the same light colors.


Process (e.g. httpd)
Application log (e.g., y.y.y.y POST... HTTP/1.1 200 ...
Socket (e.g., z.z.z.z)
Process (e.g., postgresql)
Application log (e.g., SELECT * FROM users WHERE ...)

(b) An example provenance graph of Omega-Log [44] for comparison.

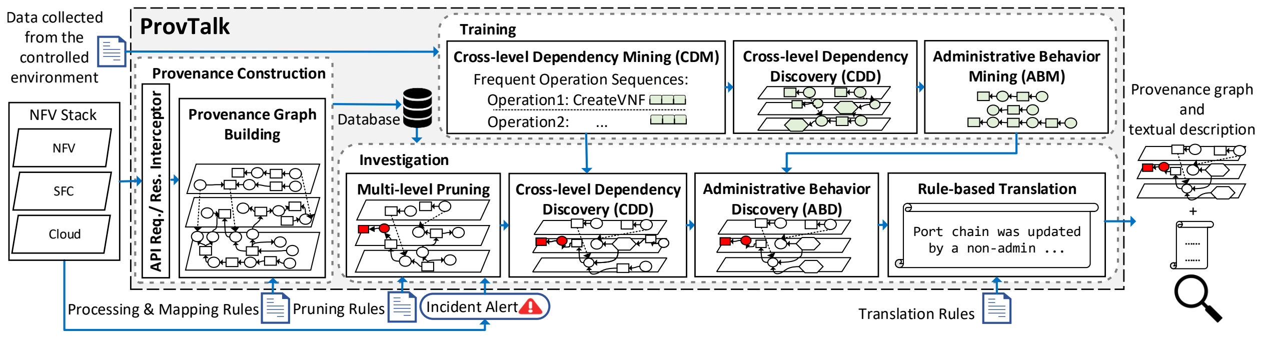

App (e.g., fwd)
Control plane dataobject (e.g., InboundPacket)
App function call (e.g., PacketContext inPacket())

(c) An example provenance graph of ProvSDN [112] for comparison.

Figure 24: An example output of ProvTalk (a), and comparing it to existing works [44, 112] to highlight the different multi-level nature (b) and (c).

4.3 ProvTalk

In this section, we detail different modules of our approach. Fig. 25 shows an overview of ProvTalk consisting of three main stages: provenance construction, training and investigation.



Data collected from the controlled environment
ProvTalk
Training
Cross-level Dependency Mining (CDM)
Frequent Operation Sequences:
Operation1: CreateVNF
Operation2: ...
Cross-level Dependency Discovery (CDD)
Administrative Behavior Mining (ABM)
Provenance Construction
NFV Stack
NFV
SFC
Cloud
API Req./ Res. Interceptor
Provenance Graph Building
Database
Investigation
Multi-level Pruning
Cross-level Dependency Discovery (CDD)
Administrative Behavior Discovery (ABD)
Rule-based Translation
Port chain was updated by a non-admin ...
Provenance graph and textual description
+
Processing & Mapping Rules
Pruning Rules
Incident Alert
Translation Rules

Figure 25: The overview of ProvTalk.

4.4 Provenance Construction

In this section, we define our NFV provenance model, and describe the provenance construction module.

4.4.1 NFV Provenance Model

We define a platform-independent provenance model based on the standard specification PROV-DM [18]. Table 8 shows a summary of nodes defined in our provenance model, their related NFV concepts and their mapping into the PROV-DM model. Subtypes refine nodes classification with respect to NFV concepts. The example provenance graph in Fig. 24a follows this model to show two types of nodes: *entities* and *activities*. Entities (shown as ovals) represent virtual resources (e.g., VM_{mal}), and activities (shown as boxes) represent management operations (e.g., *CreateVM*). To avoid cycles, we adopt the node versioning method as in [75, 92], where a new version of an entity is created if an operation affects its represented resource. For example, Fig. 24a shows that a new node representing $PortChain_x$ (i.e., the node $\langle PortChain_x, Version1\rangle$) is created after it gets updated by *Update-Port-Chain* operation. Directed edges denote the dependency between an operation and its generated or used resources. For instance, the edge from *Update-Port-Chain* to $PortPairGroup_{mal}$ shows that this operation uses $PortPairGroup_{mal}$ to update $PortChain_x$. Finally, we represent *cross-level dependencies* by edges labeled as *MappedTo*, which connect the entities related to the same resource at different levels, e.g., the edge between VNF_{ids} and VM_b in Fig. 24a.

Table 8: Mapping of the common concepts in NFV stack to the PROV-DM Model.

NFV Concept	**Description**	**PROV-DM**	**Subtype**
NFV Client	Customers of network services with specific privileges and service requirements.	Agent	NFV user admin, other tenants.
Cloud Tenant	A group of users owning an isolated set of virtual resources.	Agent	Tenant Admin, other tenants
NFV Client Operation	Management API calls for updating the life-cycle or state of network services.	Activity	Create-VNF, Update-VNFFG, etc
Cloud Provider Operation	Management API calls for updating the life-cycle or state of virtual resources.	Activity	Create-VM, Update-Port, etc
Network service component	The states of individual or chain of services such as IDS.	Entity	VNF, VNFFG, etc.
Network service configuration	The states of a deployed network service configuration, e.g., virtual firewall rules.	Entity	VNF descriptors, etc.
Cloud Resource	The states of a virtual infrastructure resource, e.g., a running/stopped VM.	Entity	VMs, virtual ports, etc.
Cloud Resource Configuration	The states of a virtual infrastructure configuration, e.g., VM virtual hardware.	Entity	Security groups, Flavors, etc.
Input for changing configurations	An input data causing a change to the configuration state.	Entity	Security group rules, etc.

4.4.2 Building the Provenance Graph

To capture all operations affecting virtual resources, we deploy our event interception mechanism as middlewares [107, 106] attached to managerial services at all levels of the NFV stack. These services include but are not limited to networking, compute, storage, image and NFV orchestration services. In Section 4.9.4, we show that intercepting all management API calls through the deployment of our middlewares is enough to satisfy the *completeness* property. Moreover, to process the intercepted API calls, we define a set of parsing and typing rules according to cloud and NFV documentations [90, 82, 87]. Specifically, the information required for building the provenance graph is embedded in the intercepted API calls. Therefore, upon intercepting each API call and based on the provided rules, ProvTalk parses the fields of that API call, identifies the type of the requested operation (e.g., *CreateVNF*), determines the affected virtual resources, and the ID of the user issuing that API call. Next, it creates nodes representing the operation and the affected resources with edges capturing their dependencies. ProvTalk also stores the extracted information (e.g. user ID) and the time of interception as node attributes. Then, it appends the created nodes and edges to the provenance graph stored in the backend graph database.

Additionally, to capture the cross-level dependencies between resources, we provide ProvTalk with a set of mapping rules. Upon the creation of each NFV-level resource, NFV platforms automatically store the ID of that resource and its lower-level associated resource in specific cells of the platform databases. Therefore, we define rules specifying the queries for extracting the IDs of those resources and creating *MappedTo* edges between their nodes. For instance, the ID and associated virtual service of a VM are stored in two columns in the same row of table *nova_instances*. Accordingly, we define the rule “Upon the interception of *CreateVNF* operations, ProvTalk should issue a query to nova_instances table to extract VM-ID from the same row storing the ID of the created VNF".

4.5 Multi-level Pruning

The provenance graph may include a large number of nodes and edges that are irrelevant to the target incident. Most of the existing pruning techniques are designed for single-level provenance models [106, 117, 92], and they remove irrelevant nodes according to the analyst’s provided pruning criteria. However, in the specific context of multi-level NFV environments, this approach has the limitation of identifying and pruning irrelevant nodes only at the same level as where the target incident is detected. In other words, those techniques do not factor in the dependencies between provenance graphs captured at different levels, which can be useful in identifying the potentially irrelevant nodes across different levels. The analyst could certainly provide additional pruning criteria for identifying irrelevant nodes at every level. However, this requires more effort from the analyst, and it also assumes he/she has a good understanding about all levels of the NFV stack and corresponding security assumptions. The reliance on such assumptions may make the pruning error-prone and result in pruning nodes that are indeed relevant to the attack.

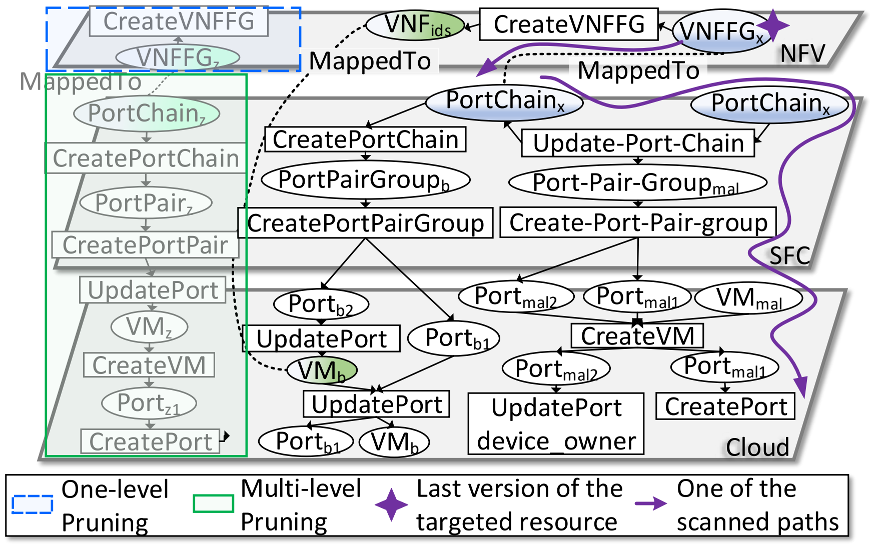


Figure 26: Excerpt of the provenance graph related to the incident discussed in Section 4.2.2 contrasting the effectiveness of one-level [106] and multi-level pruning.

To address those issues, we propose a multi-level pruning mechanism to automatically identify and keep the potentially relevant nodes across different levels by leveraging cross-level dependencies (Section 4.4.1). Specifically, after labeling all the potentially relevant nodes at the same level as the target resource (i.e., the resource associated with the incident), ProvTalk passes the label assigned to each resource to its corresponding resources at other levels via the cross-level dependencies (*MappedTo* edges). Next, ProvTalk further follows all paths to pass the labels to all the reachable nodes. To further narrow down the analysis, ProvTalk allows the analyst to specify some additional constraints about the nodes passing the labels. Finally, ProvTalk discards nodes that are not labeled as they are not reachable from the target resource.

Example 1. Fig. 26 shows an excerpt of the provenance graph in our motivating example (Section 4.2.2) contrasting one-level pruning (e.g., [106]) with multi-level pruning. Both approaches first identify the target resource $VNFFG_x$ (starred node), assign it a label and pass it to all reachable nodes meeting the provided criteria. Then, all non-labeled nodes are pruned. For the one-level pruning, as the cross-level dependencies are not considered, the label is only passed to all reachable nodes at the same level (e.g. VNF_{ids}, *CreateVNF*). Thus, only the group of nodes at the NFV-level (inside dashed blue box) are pruned, leaving the graph at other levels with potentially irrelevant nodes. In contrast, ProvTalk leverages cross-level dependencies (i.e., *MappedTo* edges) to pass the label from the target resource to nodes at lower levels (e.g., $PortChain_x$ and the path specified by an arrow). Thus, the irrelevant nodes inside the green box can also be identified and pruned.

4.6 Aggregation

The pruned provenance graph may still include a large amount of redundant information. However, pruning those is not a viable option as they may contain valuable information about the root cause. For example, some operations such as *CreatePortPair* and *CreatePortPairGroup* (hatched box in Fig. 24a) are frequently issued by cloud admins as a part of routine maintenance tasks. However, as explained in Section 4.2.2, they may also be part of the attack steps (e.g., operations issued by the attacker to insert a malicious VM into a port chain). Hence, if we hypothetically remove those operations due to their redundancy, the analyst would fail to pinpoint the root cause. Additionally, there is no systematic way of associating high-level semantics to those frequent operations, which could make the provenance graph easier to understand. For instance, the analyst cannot identify the cloud-level operations in Fig. 23 (groups of blue and green nodes) that were automatically triggered after each NFV-level operation *CreateVNF*.

Therefore, we propose an aggregation-based solution that *visually* aggregates the nodes corresponding to such operations into a compound node labeled with the corresponding NFV-level operation or administrative routine. Our aggregation technique is designed to be fully reversible such that each compound node can be easily expanded to show the original nodes, which allows the analyst to easily recover the potentially useful details of the aggregated nodes (hexagons and hatched box in Fig. 24a). Our approach consists of two mining-based schemes for *cross-level operations* and *administrative tasks operations*, which involve training and investigation stages. Similar to most approaches in this area (e.g., [39]), we collect training data from a controlled environment to ensure there is no involvement of malicious actors.

4.6.1 Cross-level Aggregation

The sequence of lower-level operations automatically triggered after an NFV-level operation are generally fixed, and thus frequently appear in the provenance graph causing redundancy. To avoid this, ProvTalk leverages a mining-based approach to model such sequences, and then applies the model to identify and aggregate the lower-level nodes corresponding to each NFV-level operation.

Cross-level Dependency Modeling (CDM). Since lower-level operations are generally triggered shortly after their corresponding NFV-level operations, we leverage this intuition to build a model of those lower-level operations generated within a small time interval. To this end, our API interceptors log operations triggered at all levels (i.e., NFV, SFC, and cloud) with a *timestamp* indicating the time when ProvTalk intercepts each operation. Next, based on those timestamps, ProvTalk extracts a sequence of lower-level operations triggered within t_{CDM} seconds after each logged NFV-level operation. The analyst may determine the interval t_{CDM} based on studies of the NFV platform (e.g., computational power).

However, since there may be many operations issued at almost the same time in a real-world NFV environment, the extracted sequences may include irrelevant lower-level operations (e.g., the operations triggered by other NFV-level operations). To address this issue, we model relevant operations by deriving frequent patterns of lower-level operations. Specifically, for each NFV-level operation, we feed the extracted sequences to our sequential pattern mining algorithm (an efficient self-supervised method for discovering the frequent patterns of ordered items in a controlled environment), which then outputs the list of mined patterns with their frequencies (i.e., support [116]). Finally, we identify the most frequent patterns related to each NFV-level operation, which are provided to the CDD module during the investigation stage.

Example 1. Fig. 27 depicts modeling the lower-level management operations triggered by the NFV-level operation, *CreateVNF*. ProvTalk extracts sequences of lower-level operations logged shortly after each *CreateVNF* (shown by rows of the *Sequences* table). We show two example scenarios causing different extracted sequences corresponding to the *CreateVNF* operation. Scenario1 describes cases where a single user issues a *CreateVNF* operation. Scenario2 describes cases where two NFV-level operations, *CreateVNF* and *DeleteVNF*, are issued at approximately the same time, and thus the extracted sequence includes an irrelevant operation, *DeleteVM* (triggered by *DeleteVNF*). To derive the operations triggered by *CreateVNF*, ProvTalk retrieves the most frequently observed patterns in *Sequences* table, which yields [*CreatePort*, *CreatePort*, ...] with the support value of 60%.

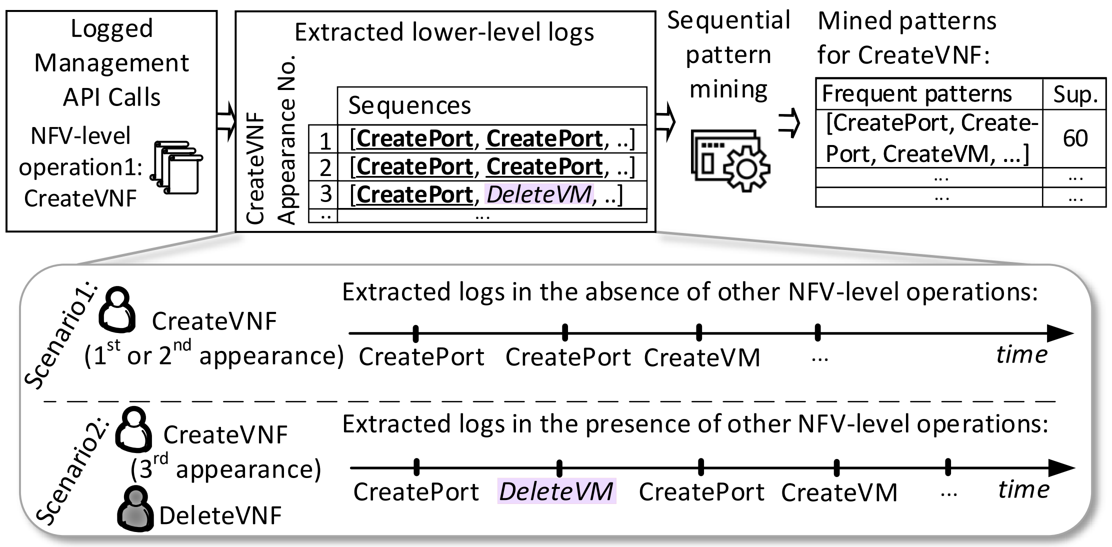
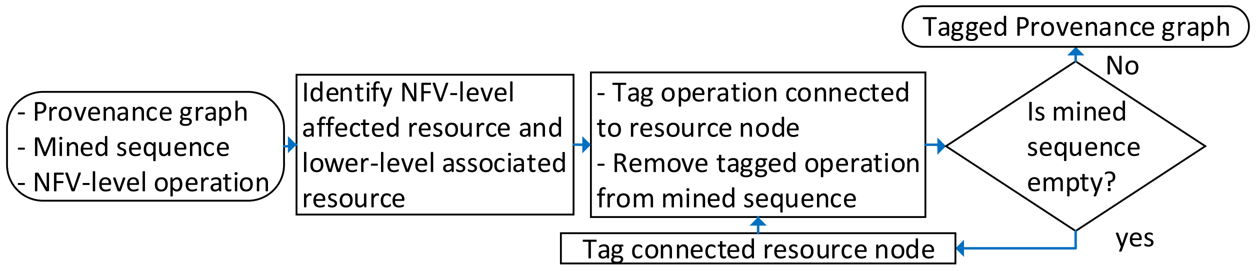

Logged Management API Calls
NFV-level operation1: CreateVNF
Extracted lower-level logs
CreateVNF Appearance No.
Sequences
1 [CreatePort, CreatePort, ..]
2 [CreatePort, CreatePort, ..]
3 [CreatePort, DeleteVM, ..]
Sequential pattern mining
Mined patterns for CreateVNF:
Frequent patterns
Sup.
[CreatePort, Create-Port, CreateVM, ...]
60
Scenario1:
CreateVNF (1st or 2nd appearance)
Extracted logs in the absence of other NFV-level operations:
CreatePort CreatePort CreateVM ...
time
Scenario2:
CreateVNF (3rd appearance)
DeleteVNF
Extracted logs in the presence of other NFV-level operations:
CreatePort DeleteVM CreatePort CreateVM ...
time

Figure 27: An example of cross-level dependency mining for the NFV-level operation, *CreateVNF* (top); Scenarios leading to different extracted sequences of lower-level operations corresponding to the *CreateVNF* operation (bottom).

Cross-level Dependency Discovery (CDD). During the investigation stage, CDD identifies and aggregates the nodes related to the mined operations corresponding to the same NFV-level operation (e.g., *CreateVNF*). This can be challenging, since there usually exist many nodes representing the same type of operation (e.g., several *CreatePort* nodes in Fig. 23). Our key insight is that, since all the triggered operations correspond to the same NFV-level operation, we can expect some dependency among them. Additionally, due to cross-level dependencies between resources, if an operation affects a resource at the NFV-level, a triggered operation will affect its associated lower-level resource. Based on such intuition, Fig. 28 shows how CDD works. First, CDD identifies the node representing the resource affected by an NFV-level operation (e.g., a created *VNF* at NFV-level) and its lower-level associated resource connected by a *MappedTo* edge (e.g., its corresponding *VM* at cloud-level). Next, to start an iteration, it tags the operation node connected to the resource node, removes that operation from the mined sequence, then tags the next connected resource node on the path. The iteration stops once the sequence is empty. Finally, CDD visually aggregates the tagged nodes into a compound node labeled as the corresponding NFV-level operation.



- Provenance graph
- Mined sequence
- NFV-level operation
Identify NFV-level affected resource and lower-level associated resource
- Tag operation connected to resource node
- Remove tagged operation from mined sequence
Is mined sequence empty?
No
Tagged Provenance graph
yes
Tag connected resource node

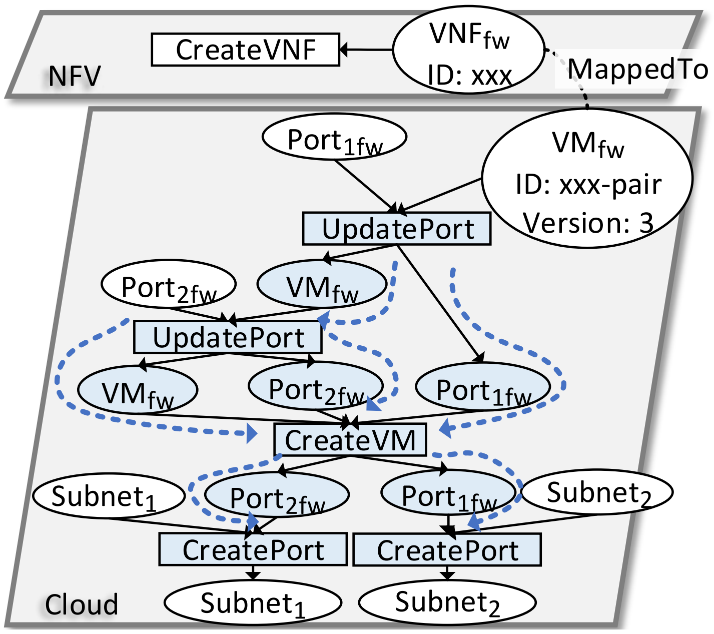
Figure 28: Identifying the cross-level dependencies.

To provide additional detail for our cross-level dependency discovery, we provide Algorithm 1, which elaborates the steps of the CDD module. For every NFV-level operation, CDD identifies and tags the nodes representing its affected resource, as well as the lower-level node connected through a *MappedTo* edge. It also tags the node representing the operation connected to the lower-level tagged node and removes that operation from the mined sequence (Line 1-3). Then, it starts an iteration over the remaining mined operations where it searches for paths of the type *(op2)←(res)←(op1)*, where *op1* is a tagged node and the operation type stored at node *op2* is in the mined sequence. It tags *res* and *op2* nodes, and removes *op2* from the mined sequence (Line 8-9). At the end of the iteration, based on the pre-specified threshold values and the number of operations remaining in the sequence, CDD aggregates the tagged nodes while labeling them with either the corresponding NFV-level operation (line 10-11), or adding *partially-mismatched* prefix to the label (line 12-13), or it does not aggregate them (line 14-15). Note that the analyst can configure the threshold values based on their platform. For instance, environments with a higher variety of services would have more various sets of lower-level operations, and therefore, he/she should provide higher threshold values in those cases.

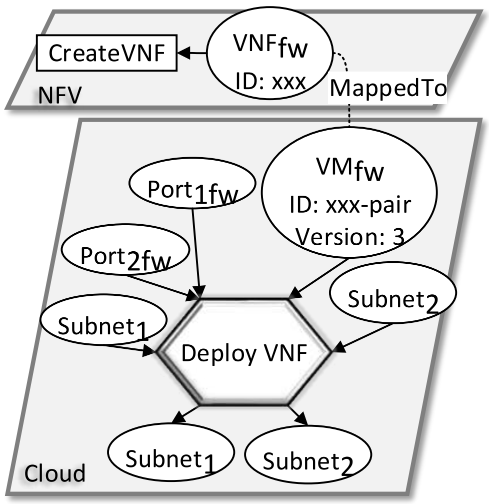
Algorithm 1 Cross-level Dependency Discovery

```
    Inputs:
        graph ← Multi-level Provenance Graph
        MinedOps ← Mined Sequences of Operations
        nfvAPIs ← NFV API Calls
        $Th_l$, $Th_h$ ← High and Low Threshold Values
    Outputs:
        Provenance graph with aggregated nodes
 1: foreach  $nfvAPI_i$ ∈ nfvAPIs  do
         %the resource and operation connected with MappedTo edge
 2:     First_Operations,graph ← MapTagger(graph, $nfvAPI_i$)
 3:     LeftOps ← OperationRemover(MinedOps, first_Operations)
 4:     while  iteration < MinedOps_len  do
 5:         if  iteration > MinedOps_len then
 6:             break
 7:         else
                %tagging other connected operations and resources
 8:             graph, FoundOperations ← Tagger(graph, LeftOps)
 9:             LeftOps ← OperationRemover(LeftOps, FoundOperations)
                %finalizing or removing tags from aggregation candidates
10:     if  LeftOps_len < Mined_len*$Th_l$ then
11:         graph←Aggregator(graph)
12:     if  LeftOps_len ∈ [MinedOps_len*$Th_l$, MinedOps_len*$Th_h$] then
13:         graph ← MismatchAggregator(graph)
14:     if  LeftOps_len> MinedOps_len*$Th_h$ then
15:         graph←TagUndoer(graph)
16: return graph
```

Example 2. Fig. 29 depicts an example aggregation related to *CreateVNF* operation. CDD identifies the lower-level node $\langle VM_{fw}, Version\ 3\rangle$ associated with VNF_{fw}. Next, it identifies the nodes representing the operations triggered by *CreateVNF* and their affected resources (Fig. 29a). The identified nodes are visually aggregated into a compound node (the hexagon in Fig. 29b), which is labeled as *DeployVNF*[1].



(a)



(b)

Figure 29: Example of cross-level dependency discovery (a) before and (b) after aggregation.

4.6.2 Administrative Behavior Aggregation

To further ease the interpretation, we aggregate the routine administrative operations (e.g., maintenance tasks) repetitively appearing in the provenance graph. ProvTalk mines the frequent sequences representing the paths (that are not aggregated by CDD) in the training stage, and aggregates the paths matching the mined sequences in the investigation stage.

Administrative Behavior Modeling (ABM). This module builds a model of routine administrative behavior based on frequent paths and using sequential pattern mining [116]. Fig. 30 shows the steps of our ABM module: 1) ABM retrieves all paths with the length of at most $l_{routine}$.

[1]We use the label *DeployVNF*, instead of *CreateVNF*, to make referring to compound nodes easier in the report.

The analyst can adjust $l_{routine}$ based on the requirements of the investigated platform, e.g., the regularity of life-cycle management of resources. 2) ABM converts the retrieved paths into string sequences. Our intuition is that nodes compose a path in a similar way that items compose an ordered sequence. Formally, a causal path can be translated into a sequence of items $[f(res\_node_i), f(op\_node_i), f(res\_node_{i+1}), ...]$ where f is the function for obtaining the string representation of a node. In this work, we use the resource or operation type attribute as the string representation of each node. For example, the path *(Port$_1$)←(CreateVM)←(VM$_1$)←(StartVM)* is converted into the following sequence: [*Port*, *CreateVM*, *VM*, *StartVM*]. 3) Finally, we apply the sequential pattern mining algorithm BIDE [116] to identify the most frequent patterns which are used by ABD during the investigation stage.

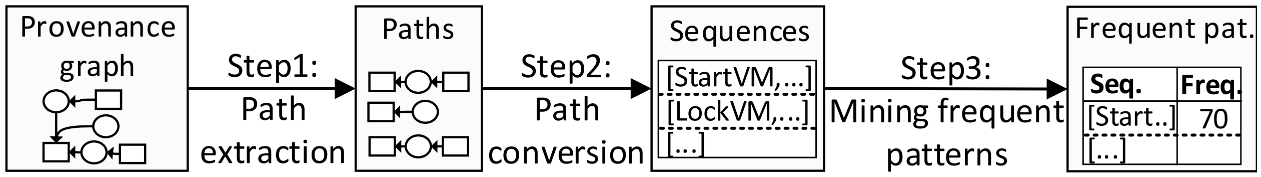
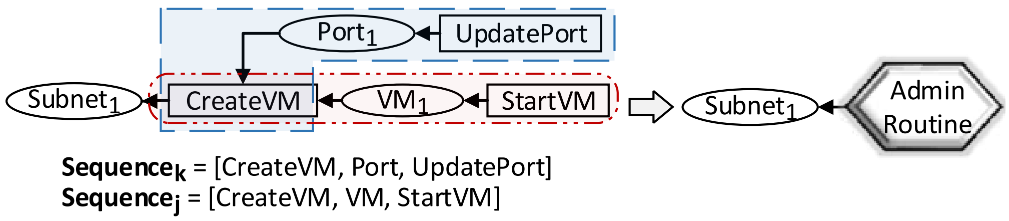


Figure 30: Steps of ABM module.

Administrative Behavior Discovery (ABD). This module identifies and aggregates the paths with a corresponding sequence that matches the patterns mined by ABM. Specifically, ABD retrieves all paths with the length of at most $l_{routine}$. Next, it converts those paths into sequences of string elements as described in the previous step (the ABM module), while it also captures the IDs (a unique number automatically assigned to each node by the graph database) of the consisting nodes. Then, it identifies the sequences that are observed among the frequent patterns mined by ABM. If a matching sequence is identified, the ABD module finds the nodes corresponding to that sequence using their unique IDs and aggregates them into a single *Admin_Routine* compound node. Moreover, to increase the number of aggregated nodes represented by each compound node (i.e., the reduction power of our scheme), we merge *Admin_Routine* nodes with common aggregated nodes.

Example 3. Fig. 31 shows an example of administrative behavior aggregation. As we can see on the left, two paths are converted into sequences $Sequence_k$ and $Sequence_j$, and are initially aggregated into the blue and red shaded compound nodes. However, due to their common aggregated node, ABD merges them into one single compound node (right side).



Port1
UpdatePort
Subnet1
CreateVM
VM1
StartVM
Subnet1
Admin Routine
Sequencek = [CreateVM, Port, UpdatePort]
Sequencej = [CreateVM, VM, StartVM]

Figure 31: Example of administrative behavior aggregation and merging compound nodes.

In Fig. 24a, we have shown our motivating example provenance graph after applying both aggregation schemes. As we can see, the resulted provenance graph is significantly smaller and more interpretable with the assigned labels.

4.7 Rule-based Translation

To further enhance the interpretability of the provenance graph and facilitate the analysis, we propose a rule-based technique to translate the captured information into a human-readable text. As we demonstrate in Section 4.9.5, the generated text can provide useful guidance to the analyst in investigating the provenance graph and identifying the root cause. The analysts may also take advantage of the generated text for filing a report describing the result of their investigations. To this end, ProvTalk first backtracks from the node corresponding to the target resource (e.g., the node $VNFFG_x$ in Fig. 24a) to retrieve all paths connected to this node. ProvTalk also allows the analyst to specify a time interval so that only paths generated during that time will be translated. Next, it extracts the information captured in each path, and generates a textual description reflecting those information and the incident alert.

As an example, Fig. 32 shows the automatically generated description of the highlighted path in Fig. 24a. The generated text is organized in three paragraphs: the first paragraph reflects the information extracted from the incident alert as well as the number of operations represented in the path. The second paragraph details the information extracted from the path, and the last paragraph includes the information necessary for identifying the described path in the provenance graph.

```
By the detection time 21-01-06 11:44:07.769, there are 6 operations performed in
the specified time interval 0:00:15.454 hours corresponding to the target entity
VNFFGx created by admin user using 2 VNFs.
User 12ddf created PORTmal2 at 07:10:03.403. He updated device_owner
Portmal2 at 07:10:05.123. And  created VMmal using that port(s) after 0:00:0.878
hours. Then, he created PortPairmal using that port(s) after 0:00:03.202 hours.
And created PortPairGroupmal using that PortPair(s). He updated PortChainx,
using that PortPairGroup(s) after 0:00:02.239 Hours.
More details can be found in the provenance graph following this node path [215
- 211 - 208 - 207 - 204 - 202]. Node(s) (UpdatePortChain-ID: 215) worth a closer
look: nonAdmin user (ID: 12ddf) modifies admin (ID: 53atb) modified resource(s).
```

Figure 32: Example of auto-generated textual description.

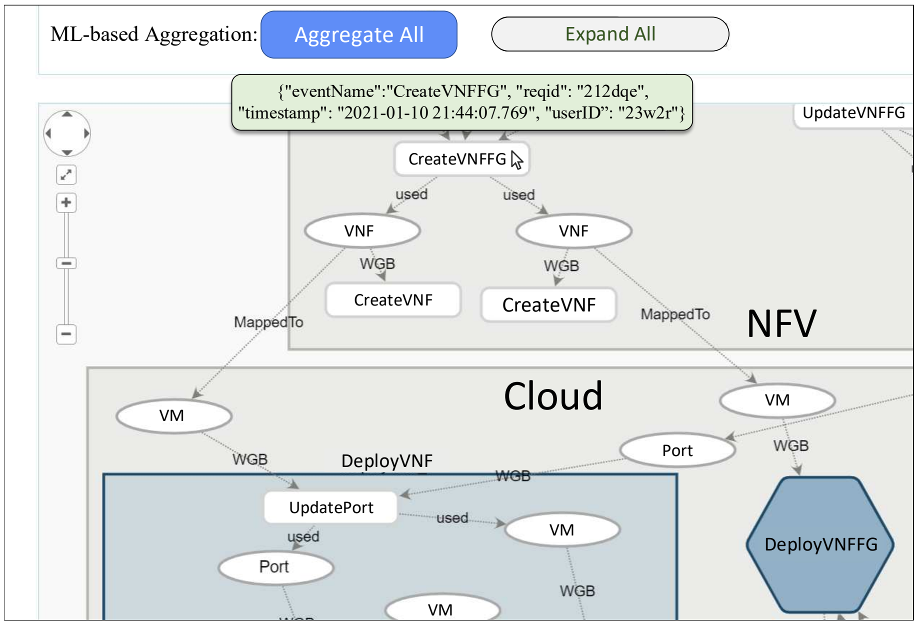
Path Translation. ProvTalk automatically follows graph paths to extract and include the following four node attributes in sentences of the second paragraph): 1) *user*: ProvTalk treats the user issuing an API call as the *subject* of the sentence, and identifies it by the user ID attribute stored in operation nodes. 2) *operations*: ProvTalk treats operation type node attribute as the *verb* of the sentence. 3) *resources*: resources affected by each operation are treated as *objects*, and are identified by the entities from which there is an edge pointing to an activity (Section 4.4.1). 4) *timestamps*: ProvTalk includes the timestamp attribute (stored in operation nodes) as the *propositional phrase* in the sentence. Additionally, to smooth the transition between sentences, it uses pronouns and *transitional words*. ProvTalk also applies pre-defined sentence templates to automatically form the descriptions. To support aggregation, analysts can configure ProvTalk to treat the label of compound nodes as the *verb* of generated sentences.

To enable retrieving the information not already reflected in the generated text, the third paragraph describes the unique IDs of the corresponding nodes, using which the analyst can map the generated text back to the provenance graph. Additionally, ProvTalk can be configured to include specific parts of the extracted information that may deserve more attention.

4.8 Implementation

We implement ProvTalk in a testbed based on Tacker [88], SFC [87] and OpenStack [83] (a popular platform supporting NFV for telecommunication service providers [113, 84]). We note that only our API interception mechanism is platform-specific (i.e., OpenStack/Tacker), while the modular design of our approach makes it readily adaptable to other multi-level virtualized platforms (see Section 4.10 for detailed discussion). To intercept provenance metadata from the REST API calls issued to different services (i.e., Tacker, SFC, and Openstack services, e.g., Nova and Neutron), we implement our provenance construction module as Python WSGI middleware [120, 60], which stores the provenance graph in Neo4j [80] database, and uses Cypher language [78] to query the database. We implement pruning and aggregation modules in Python, and use BIDE algorithm [116] to mine frequent sequences. We set the interval t_{CDM} to 15 seconds (Section 4.6.1) and $l_{routine}$ to 10 (Section 4.6.2). We also use the Th_h=0.8 and Th_l=0.5 as the thresholds in our aggregation. Our translation module exports the provenance graphs into JSON format [79], and uses SimpleNLG Python realiser [37] for generating sentences. To visualize the provenance graphs and enable the interaction with ProvTalk, we provide a frontend graphical user interface. We use Cytoscape [28] to visualize the provenance graph, and support the aggregation and expansion of compound nodes. We show a screenshot of our interface with the magnified excerpt of a provenance graph in Fig. 33. A brief summary of recent incident alerts is displayed in our interface (not shown in Fig. 33). By selecting each incident, users can see the provenance graph corresponding to each incident, and then invoke the pruning and aggregation options. Users can click on the compound nodes (blue hexagon in Fig 33) to expand them and visualize the aggregated nodes (appeared in the blue rectangle). Users can also examine the information about the operations and affected resources (the green box in Fig. 33), by hovering over their corresponding nodes.

To demonstrate that our approach can potentially be applied to environments where intercepting



{"eventName":"CreateVNFFG", "reqid": "212dqe", "timestamp": "2021-01-10 21:44:07.769", "userID": "23w2r"}

Figure 33: Example screenshot of ProvTalk showing the aggregated and expanded cloud-level nodes and the information shown while hovering a node.

API calls may not be feasible, we implement a log processor for extending ProvTalk to work with infrastructure logs (e.g., logs generated by *Neutron-Server* services by default). A challenge is that some details of the API calls are not captured by their corresponding log entries [67]. For instance, *Neutron-Server* does not log all resources affected by most API calls such as the virtual subnet attached to a created port. To collect some missing details, we devise methods for automatically inferring them through correlating the log entries of different services based on the ID of resources as index. For example, by correlating *Neutron-Server* and *DHCP-agent* logs, we extract the virtual subnet that is attached to a port. We implement this method as a log processor module using Python to automatically extract and correlate our required information from different services' logs. This additional module can potentially extend the scope of application for ProvTalk to cover other virtual environments as long as there exists the logging capability.

4.9 Evaluation

To evaluate ProvTalk, we seek to answer the following questions:

RQ1: How effective is the provenance model at capturing real-world attacks in NFV environments?

RQ2: To what extent can ProvTalk reduce the size of the provenance graph? What is the effect of accuracy on the performance? How does it compare to the existing techniques?

RQ3: What is the overhead introduced by ProvTalk in terms of latency, computation and storage? How does it compare to the existing OS-level provenance techniques?

RQ4: How complete and sound is ProvTalk in terms of capturing and analysing all management API calls?

RQ5: How helpful is the enhanced interpretability in root cause analysis for real-world users?

Experimental Setup and Dataset. We run ProvTalk on an Ubuntu 18.04 server equipped with Intel Xeon Bronze 3104 CPU @1.70GHz and 128GB of RAM. We conducted our experiments based on both our testbed and a real research cloud dataset. To generate diverse sequences of operations in our dataset, we deploy 31 different types of VNFs while randomly varying their parameters (e.g., the number of virtual ports), and seven variations of VNFFGs with varying parameters (e.g., the number of VNFs per VNFFG). Table 9 shows statistics about the datasets generated in our NFV testbed.

Table 9: Statistics of our NFV testbed datasets.

	Training Datasets				**Testing Datasets**			
# of API calls (in thousands)	6	9	12	15	3	6	9	12
# of nodes (in thousands)	11	16	21	28	5	10	15	20
# of VMs (in hundreds)	6	8	10	11	3	6	9	11
# of VNFs (in hundreds)	3	5	6	6	3	4	6	7

Table 10: Attack scenarios used to evaluate the effectiveness of ProvTalk (the shaded rows indicate the incident and root cause are located at different levels).

Root Cause	**Detected Incident**	**Most Relevant Management Operations**	**Vulnerability**
Stealthy node injection into a VNFFG [110]	Unauthorized Access	Create-Port-Pair, Create-Port-Pair-Group, Update-Port-Chain	CVE-2017-2673
Bypassing anti-spoofing rules in network [107]	Unauthorized Access	Create-VNFFG, Create-Port, Create-VM, Update-Port	CVE-2015-5240
Firewall VNF misconfiguration	CPU DoS	Create-VNFFG, Update-VNFFG, Update-VNF	CVE-2017-7400
Malformed security group rule addition	Host Unavailability	Create-Security-Group, Create-Security-Group-Rule, Create-VM	CVE-2019-9735
Overlapping security group rule addition	Host Unavailability	Create-Security-Group, Create-Security-Group-Rule, Create-VM	CVE-2019-10876
Update of security group is not applied [66]	Data Leakage	Add-Security-Group, Start-VM, Delete-Security-Group-Rule	CVE-2015-7713
Neutron proper authorization failure [119]	Port Scanning	Create-Router, Create-Port, Create-VM	CVE-2014-0056
Wrong VLAN ID [21]	Data Leakage	Create-Network, Update-Network	Not specified
Failing to delete VMs in resize state	Disk DoS	Create-VM, Resize-VM, Delete-VM	CVE-2016-7498
Excessive VM creation on the same host [64]	Disk DoS	Create-VM	Not specified

4.9.1 Effectiveness

To answer RQ1, we automatically reproduce in our testbed 10 attack scenarios that involve NFV management operations (as discussed in e.g., [110, 66, 21]) via a Bash script. Table 10 summarizes those scenarios, the most relevant operations and the vulnerabilities that are exploited through the requested operations for launching the attack. We evaluate the effectiveness of ProvTalk on these attacks as we know the precise ground truth (attack steps) published in the existing works[1] and the publicly reported vulnerabilities [29]. For all the 10 attacks, we successfully trace back to the root cause of the reported incident using ProvTalk. Note that due to the novelty of NFV, very few papers exist on NFV-specific attacks or vulnerability exploits which limit our choice of attacks (although given the prevalence of NFV 5G telecommunication, we envision an extensive research on this matter in the future). We showcase the effectiveness of our approach based on four cases: two vulnerabilities in Table 10 presented in the motivating example (first and second rows) and two cases presented below (third and seventh rows). We choose those scenarios because their incidents are detected at a different level from where the root cause operations are conducted, which make the analyses more challenging.

[1]Most of these works (e.g., [110, 66]) focus on *security verification* (rather than provenance analysis), and thus we cannot directly compare our results with these solutions.

4.9.1.1 Cloud-level Alert, NFV-level Root cause

In this scenario (see Table 10, third row), a VNFFG with a virtual firewall is protecting an end-to-end network service. The analyst receives a high CPU utilization alert from VM_b. Using the provenance graph generated by ProvTalk (Fig. 34), the analyst can observe that the VM generating the alert at cloud-level (VM_b) corresponds to VNF_1 (1). He/She can also see that VNF_1 was included in $VNFFG_z$ (2). Moreover, according to the provenance graph, $VNFFG_z$ was updated by the admin for chaining a preceding virtual firewall (3). However, shortly after adding the firewall, another user changed its configurations so that it will not filter syn-flood traffic (4). As updating the configuration to allow syn-flood traffic right after the insertion of a firewall in a VNFFG is not a routine behavior and is conducted by a non-admin user, the analyst can attribute this to a potential privilege escalation by that user.

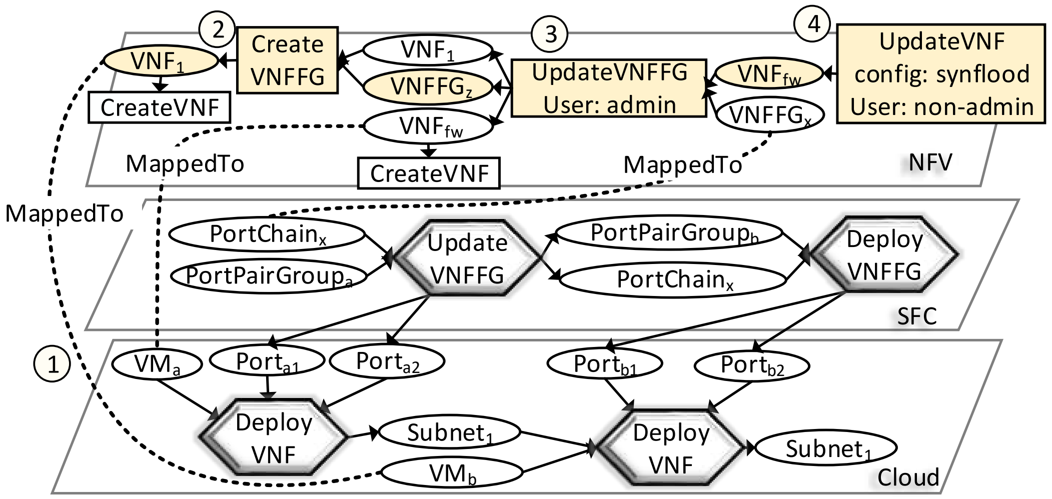


Figure 34: Root cause of the CPU DoS Identified by ProvTalk.

4.9.1.2 NFV-level Alert, Cloud-level Root Cause

In this scenario (see Table 10, seventh row), the analyst receives a port scanning alert from a virtual IDS service, VNF_{ids}. Using the multi-level provenance graph generated by ProvTalk (Fig. 35), he/she can easily identify that VNF_{ids} is associated with VM_b, which is created in $Subnet_1$ (1). He/She can also observe that an attacker from a different cloud tenant creates another port in

Subnet$_1$ attaching it to *Router$_1$* (2), and then creates a VM attached to this port (3). As this chain of operations shows that a different tenant could enter the network of the target resource, the analyst suspects that an attacker exploited a vulnerability in the NFV platform [85] to send malicious traffic to *Subnet$_1$*, which is detected by *VNF$_{ids}$*.

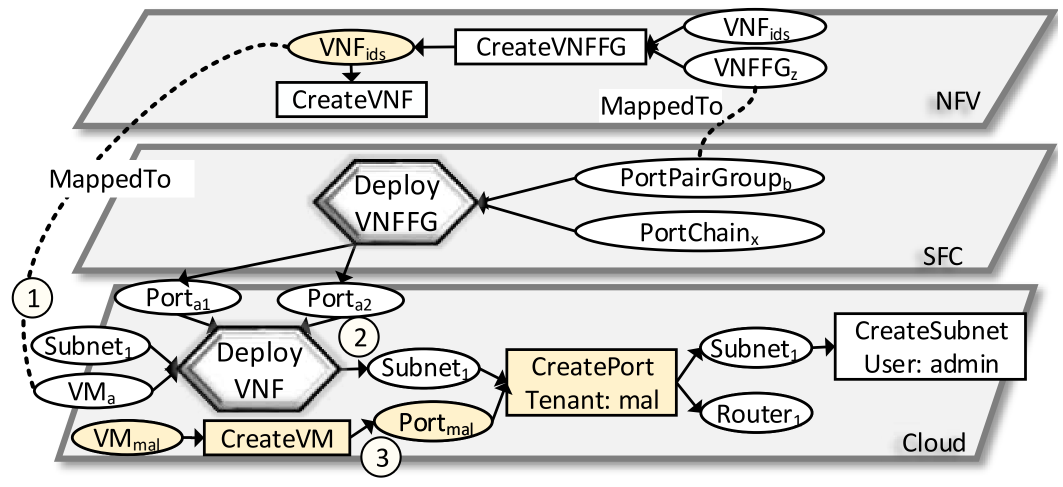
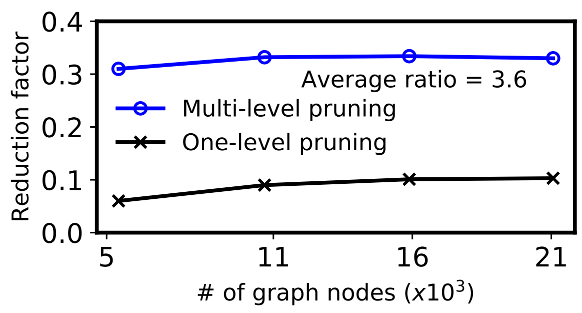


Figure 35: Root cause of port scanning identified by ProvTalk.

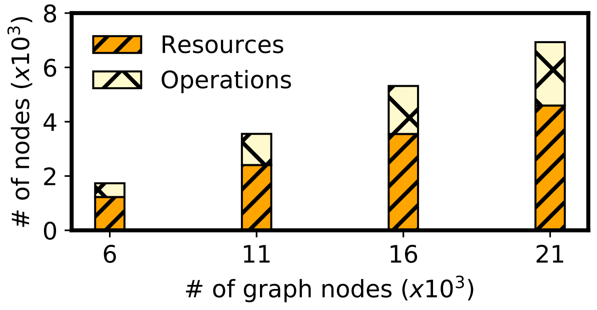
4.9.2 Graph Reduction Performance

To answer RQ2, we measure the reduction in the size of the provenance graph after applying our pruning and aggregation.

Comparing multi-level pruning with one-level pruning. To measure the effectiveness of multi-level pruning, we apply both multi-level pruning and one-level (NFV-level) pruning on provenance graphs of different sizes for the same incident. Fig. 36a shows that the reduction factor (i.e., the number of pruned nodes over that of all nodes) of multi-level pruning scheme is significantly higher (on average, by almost 3.6 times) than the one-level pruning scheme in all datasets. Furthermore, Fig. 36b shows that multi-level pruning removes a larger number of nodes representing resources than those representing operations. The reason is that most operations affect several resources at the same time, therefore, pruning an operation will automatically prune its many related resources.



(a) Multi-level vs. one-level effectiveness.

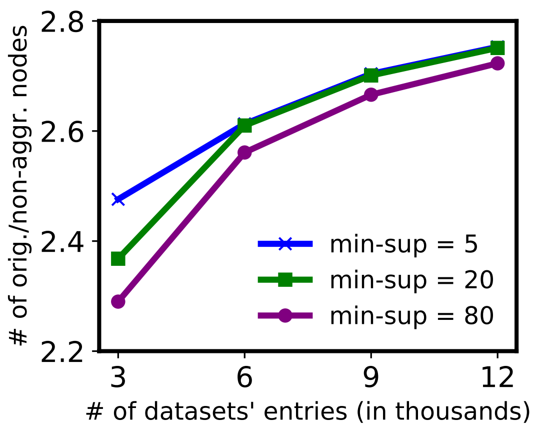


(b) Multi-level pruned resources vs. operations.

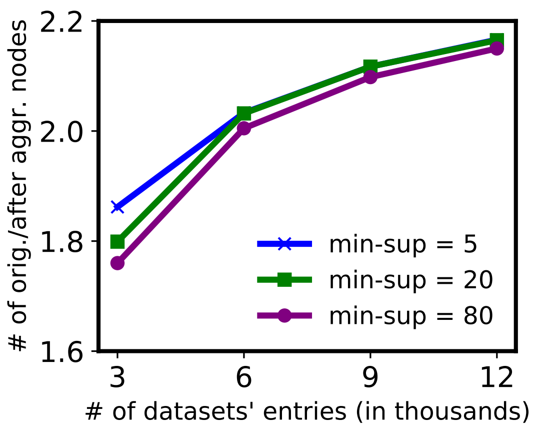
Figure 36: Evaluating the effectiveness of multi-level pruning.

Aggregation. We measure the effectiveness of our aggregation for datasets of different sizes, while varying the minimum support values (aka *min-sup*) [116]. The min-sup value indicates the minimum acceptable frequency of mined patterns in our training data, and thus, by varying the min-sup value from low to high, we can evaluate the performance for moderately to highly conservative scenarios respectively. Fig. 37a shows the ratio between the number of nodes in the original graph and that of the non-aggregated nodes. Fig. 37a shows that, on average, the number of nodes in the original provenance graph is around 2.6 times larger than the number of non-aggregated nodes and the reduction ratio increases with the size of the dataset. The reason is that, in larger provenance graphs, there usually exist more diverse sequences of operations, which increases the level of redundancy, and hence, allows for more aggregation. Moreover, smaller min-sup values lead to a lower ratio of non-aggregated nodes in smaller datasets, in contrast with larger datasets where reduction is similar for all measured values. The reason is that smaller datasets are more likely to lack the patterns mined via greater min-sup values, i.e., some of the most frequent mined patterns in the training data.

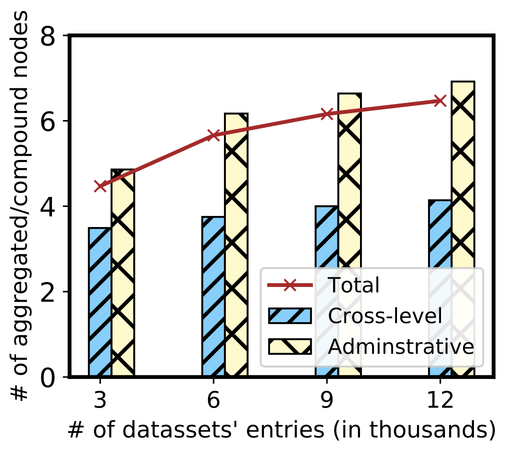
Fig. 37b shows the ratio between the number of nodes in the original provenance graph and in the one after aggregation (which consists of non-aggregated and compound nodes). On average, our aggregation schemes decrease the size of the provenance graph by half, which shows the usefulness of our approach in providing a smaller provenance graph for investigation with the added higher-level semantics. Furthermore, the ascending trend of the curve shows that, as the size of the original provenance graph grows, our aggregation schemes is more effective in terms of reducing the graph



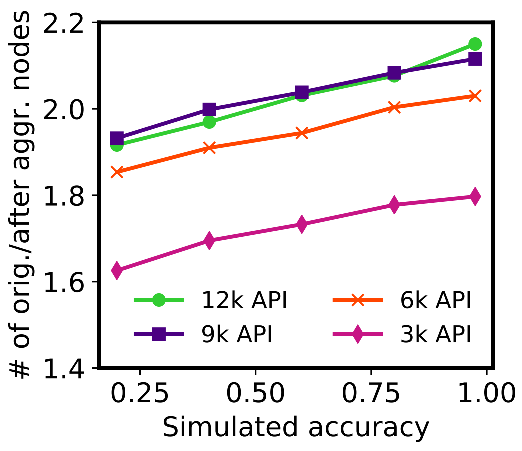
(a) Aggregated portion growth.



(b) Size reduction growth.



(c) Accuracy effect on size reduction.



(d) Aggregation power growth.

Figure 37: Evaluating the effectiveness of aggregation.

size. One reason is that in larger provenance graphs there usually exist more overlapping compound nodes subsequently aggregated into a single compound node (Section 4.6). In other words, as the size of the provenance graph grows, larger groups of nodes are aggregated into a single compound node, which leads to a greater reduction power. We can also see that lower min-sup values lead to a greater size reduction in smaller testing datasets, which is aligned with our results of Fig. 37a.

To gain more insights into the effectiveness of the aggregation, we evaluate the aggregation power of our approach (i.e., the number of aggregated nodes represented by compound nodes). Fig. 37c shows that the ratio between the total number of aggregated nodes and that of the compound nodes grows with the size of the provenance graph. On average, 5.69 aggregated nodes are represented by each compound node. To evaluate the performance of each aggregation scheme, we measure their contribution to reducing the size of the provenance graph separately. As we can see in Fig. 37c, for all datasets, the ratio between the number of compound nodes and the aggregated nodes they represent is higher for the administrative aggregation scheme. This can be partially explained by the merging we conduct on overlapping compound nodes related to the administrative behavior (Section 4.6). Note that we do not merge cross-level compound nodes, as each of those nodes corresponds to a particular NFV-level operation and are labeled accordingly. Moreover, the ratio between aggregated and compound nodes under both schemes increases with the size of the provenance graph, which shows the better performance of both schemes for larger datasets.

Effect of accuracy on the reduction power. To evaluate how much the reduction power of ProvTalk may be affected by inaccuracies of our aggregation module, we simulate different accuracy values (of the frequent pattern mining step of our aggregation module) ranging from 20% up to 97%. To do so, we first manually verify the paths that are identified to be related to cross-level dependencies or administrative behaviors in our testing datasets. Then, we configure ProvTalk to randomly select a number of validated paths and leave them non-aggregated in the provenance graph. We determine the number of these *ignored* paths according to the desired accuracy value and the total number of paths extracted from the provenance graph. Fig. 37d shows the variation in the size reduction caused by different accuracy values in our four testing datasets. On average, the lowest accuracy value of 20% decreases the reduction power by almost 0.19 times only (i.e., preserving the total reduction power of around two times). It is also worth noting that since our aggregation module provides a more compact instance of the provenance graph (rather than discarding the nodes or the attack detection), low accuracy values would only affect the reduction power without losing vital information or generating false alarms.

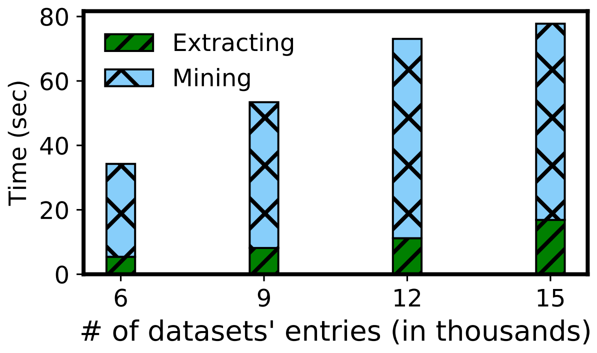
4.9.3 Efficiency

To answer RQ3, we measure the efficiency of ProvTalk based on our NFV testbed and real-world dataset.

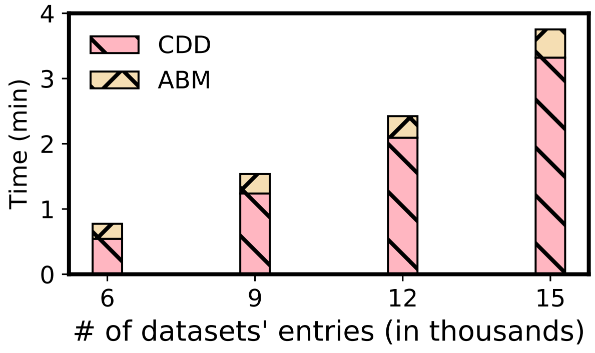
4.9.3.1 Scalability Evaluation with NFV Testbed

To evaluate our approach in environments with a large number of diverse management API calls and deployed virtual services, we run ProvTalk on datasets generated in our NFV testbed (Table 9).

Training time consumption. We measure the time required by the training stage of ProvTalk. Worthy to note that this is a one-time cost since the lower-level operations triggered by NFV-level API calls and administrative tasks do not change very frequently. Fig. 38a shows the time required by the CDM module for extracting the logged lower-level operations and running sequential pattern mining algorithm over the extracted operations. While the time required by both steps grows almost linearly with the size of the dataset, the total time does not exceed 80 seconds for the largest dataset. Fig. 38b depicts the time required by the CDD and ABM modules. Although the required time increases with the size of the datasets, it remains under four minutes in total for the largest dataset.



(a) CDM module steps delay.

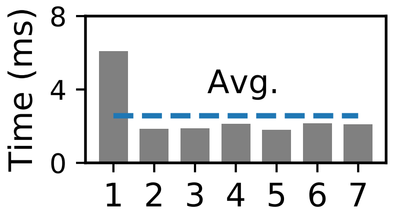


(b) CDD and ABM modules delay.

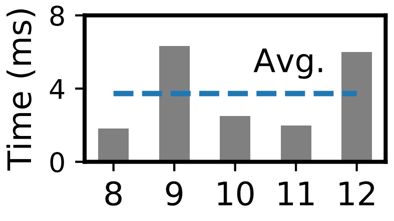
Figure 38: Evaluating the training stage overhead.

Runtime delay. We measure the delay caused by ProvTalk in the runtime execution of NFV management operations. Fig. 39 shows that, on average, ProvTalk adds around a two-millisecond delay for logging the information of most cloud-level operations. Longer delays (around eight milliseconds) mostly correspond to operations which also have a longer execution time (e.g., *CreateVM* has an execution time of above 10 seconds [65]). The average delay increases to 4 and 6 milliseconds for SFC-level and NFV-level operations (which also have a longer execution time). In summary, ProvTalk incurs a negligible overhead of around 0.04% additional delay to NFV management operations.

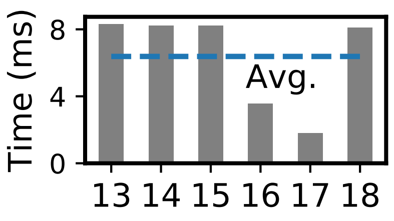
1. CreateVM	**2.** StartVM	**3.** UpdatePort	**4.** LockVM
5. LinkSubnets	**6.** CreateNet.	**7.** CreateVMPass.	**8.** CreatePortPair
9. CreatePortChain	**10.** CreatePortPairGroup	**11.** UpdatePortChain	**12.** DeletePortChain
13. CreateVNFFG	**14.** CreateVNF	**15.** DeleteVNF	**16.** UpdateVNFFG
17. UpdateVNF	**18.** DeleteVNFFG		



(a) Cloud-level.



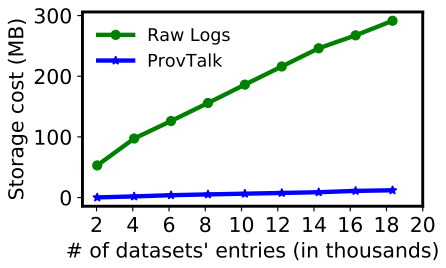
(b) SFC-level.



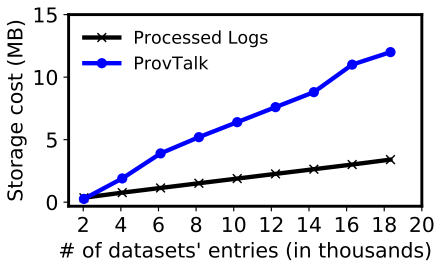
(c) NFV-level.

Figure 39: Runtime delay imposed to API calls at different levels.

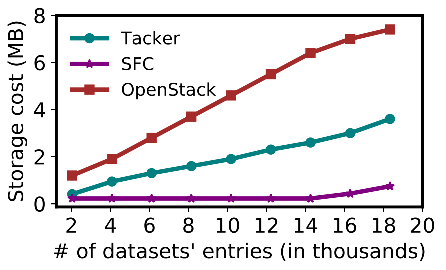
Storage cost. We also evaluate the storage consumption of ProvTalk. Fig. 40a shows that the storage required by ProvTalk remains significantly lower than the logs generated by the platform. The reason is that the generated logs contain a large amount of information (e.g., listing deployed services) that are less important for security analysis, and thus are not included in the provenance graphs. Fig. 40b compares the storage required by ProvTalk and processed logs (containing only the information that we store as the attributes of nodes). Since the processed logs do not capture the relationships between operations necessary for the analyses, the size of the provenance graphs is greater than the processed logs, while it remains around 10 megabyte in the largest dataset with 20,000 API calls. To evaluate the storage cost of ProvTalk for environments with different available services, we measure the storage required for provenance graphs that are recorded at each level separately. Fig. 40c shows that the storage related to OpenStack API calls is significantly higher than the others, which can be caused by the higher number of API calls at this level (triggered both by cloud users and the platform after NFV-level operations). On the other hand, SFC operations consume the least amount of storage, due to the limited type and number of API calls that are attributed to this level.



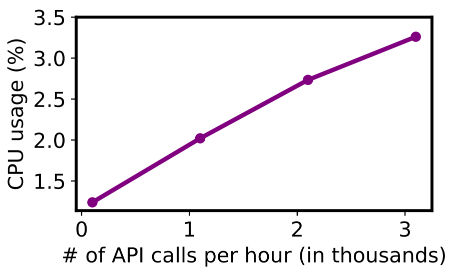
(a) PG vs. raw logs storage cost.



(b) PG vs. processed logs storage cost.



(c) per level of PG storage cost.



(d) PG construction CPU usage.

Figure 40: Evaluating the storage and computation cost of ProvTalk (PG denotes provenance graph).

CPU consumption. We measure how the rate of incoming API calls impacts the CPU usage. To simulate setups with different workloads, we vary the rate of received API calls per hour so that the time elapsed between every two consecutive API calls would be a fraction of the time elapsed between the same API calls in our real data. Fig. 40d shows that the CPU usage during the construction of the provenance graph increases almost linearly with the rate of received API calls. The rate of about 1,000 API calls per hour is comparable to the workload of our research cloud which incurs less than 1.25% CPU consumption. For enterprises with higher rates of API calls (e.g., 3,000 API calls per hour), the CPU consumption remains under 3.5%, which shows the scalability of our approach.

Comparing with OS-level provenance approaches. We also evaluate the benefit of tracking management API calls over their corresponding OS-level events. Table 11 shows the size of the OS-level provenance graph generated following each management API call as well as the provenance graph constructed by ProvTalk (two bottom rows). To build the OS-level provenance graph, we deploy a widely used open source tool, SPADE [38], in our controller host. We show an excerpt of the OS-level provenance graph generated after *CreateVNF* operation in Fig. 41. We can see that the OS-level provenance graphs would be impractically large in NFV environments (reaching millions of nodes and edges) even under a moderate workload of a research cloud with thousands of API calls issued during only a few days (Section 4.9.3.2). Additionally, in contrast with OS-level events, API management interfaces are usually accessible to a broad range of NFV/cloud customers (e.g., AWS CloudTrail) [51], which may include potentially malicious users. Therefore, by tracking management operations, ProvTalk enables an effective analysis on a vast group of security incidents in NFV, while avoiding the cost of large OS-level provenance graphs. Moreover, tracking management API calls provides a higher-level perception of changes in the NFV stack, and thus enables easier root cause identification.

Table 11: Comparing the size of OS-level provenance graphs generated following each management API call with ProvTalk.

OS-level	**CreateVNF**	**DeleteVNF**	**CreatePortChain**	**LockVM**
# of nodes	2582	2548	13065	234
# of edges	12692	10116	38585	615
Storage (mb)	3.3	2.4	9.9	0.27
ProvTalk	**CreateVNF**	**DeleteVNF**	**CreatePortChain**	**LockVM**
# of nodes	16	16	35	2
# of edges	19	19	42	2

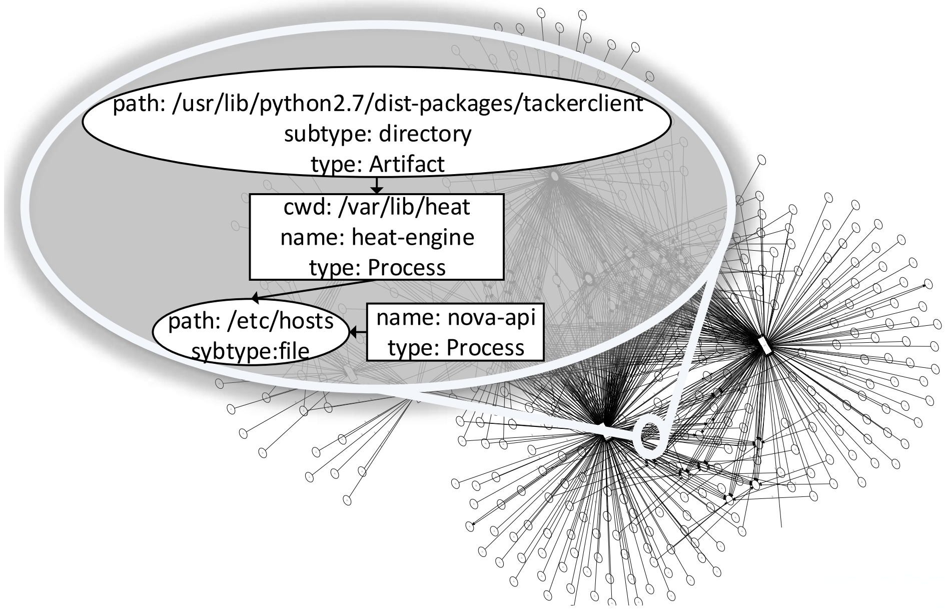


Figure 41: Excerpt of the OS-level provenance graph generated by SPADE [38] upon issuing the *CreateVNF* operation. We magnified a subgraph to illustrate an example of the relationships between processes and data objects in an NFV controller.

4.9.3.2 Experiments with Real-world Data

We evaluate the applicability of our approach by using 5 days of OpenStack logs collected from a real research cloud hosted at a major telecommunications vendor with hundreds of hosts and users. Although logs generated by the platform lack sufficient information [67], we build the provenance graph using those logs as we were not allowed to install API interceptors in this cloud (detailed in Section 4.8). Those logs consist of 1,882 API calls affecting the deployment configuration of 354 VMs. Note that the number of the extracted API calls is smaller compared to our NFV testbed dataset, due to the unavailability of Tacker-level logs in that environment. Accordingly, ProvTalk generates a provenance graph of 2,157 nodes in 99.8 seconds which consumes only 2.53 megabytes storage. Thus, ProvTalk imposes negligible storage costs in real-world virtualized environments.

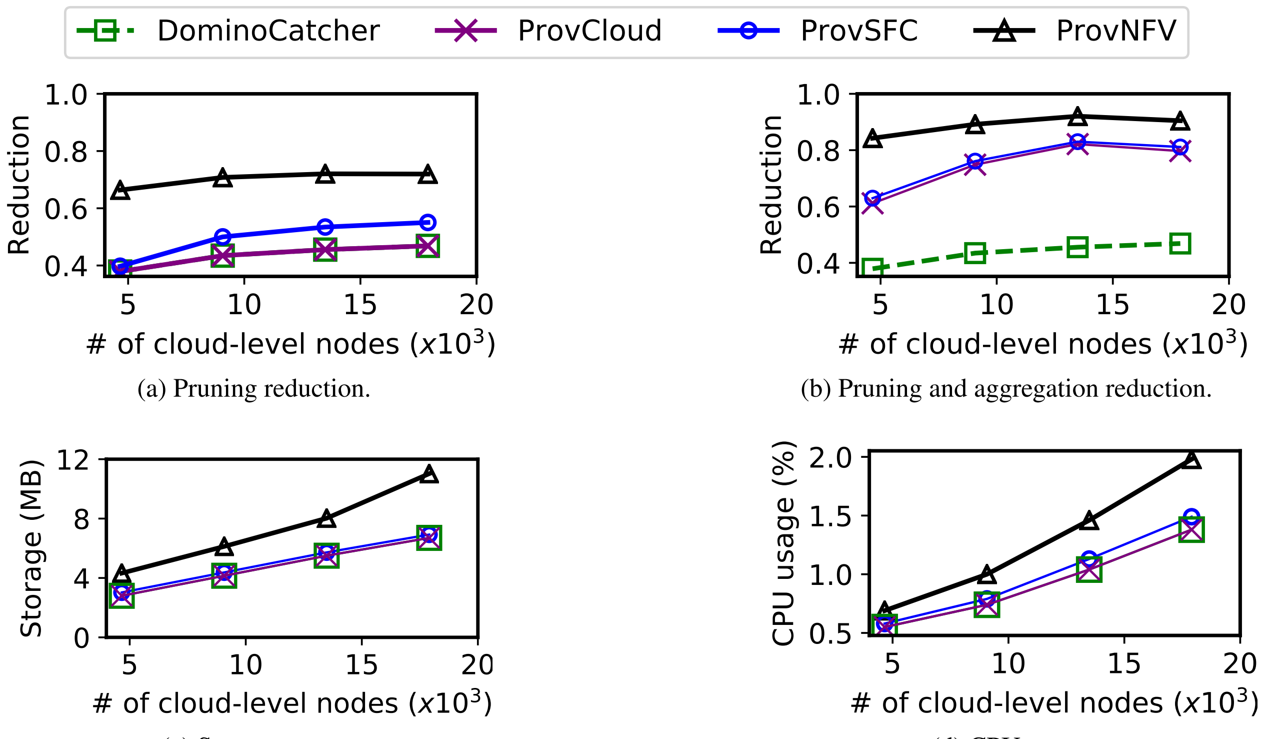
We conclude that unlike existing techniques localizing failed components (e.g., [53, 100]), ProvTalk facilitates identifying the root cause activities, while incurring reasonable costs.

4.9.3.3 Comparing with DominoCatcher

We compare the overhead and performance of ProvTalk with DominoCatcher [106]. While DominoCatcher can only support the cloud level, ProvTalk can be applied to different virtual environments (e.g., cloud only, cloud and SFC, and cloud/SFC/NFV). Therefore, while all the experiments are based on cloud-level provenance graphs (the only level captured by DominoCatcher), we have applied ProvTalk under three scenarios, i.e., cloud-level only (*ProvCloud*), considering the cross-level dependencies between cloud/SFC levels (*ProvSFC*), and cloud/SFC/NFV levels (*ProvNFV*).

Size reduction. Fig. 42a shows the reduction of cloud-level provenance graphs of our testing datasets after applying the pruning schemes of DominoCatcher and ProvTalk (under aforementioned scenarios). The reduction factor is the number of pruned nodes over the total number of the cloud-level nodes. As Fig. 42a shows, the pruning schemes of ProvCloud and DominoCatcher have a similar reduction factor as neither of them captures cross-level dependencies. In contrast, ProvSFC and ProvNFV provide around 13% and 60% further reduction, respectively, which confirms the added benefits of our multi-level pruning schemes. In addition, Fig. 42b compares the total reduction enabled by both the pruning and aggregation techniques of ProvTalk with DominoCatcher. The administrative aggregation of ProvCloud enables around 71% further reduction compared with DominoCatcher. In constrast, ProvNFV has twice the reduction factor of DominoCatcher. Note that there is a smaller reduction caused by ProvSFC. The reason is that ProvSFC first prunes the portion of cloud-level nodes that corresponds to the routine administrative behavior, and thus a smaller number of nodes will remain to be aggregated by ProvSFC.

Overhead. Fig. 42c shows that ProvNFV consumes only around 5 megabytes additional amount of storage compared with DominoCatcher. The additional storage cost increases as the number of NFV-level operations and resources grows (from 300 to 700 VNFs in our testing datasets). The storage required by DominoCatcher is similar to that by provenance graphs of ProvCloud, as both schemes capture only the dependencies between cloud-level operations. Note that ProvSFC



(a) Pruning reduction.

(b) Pruning and aggregation reduction.

(c) Storage cost.

(d) CPU usage.

Figure 42: Comparing the effectiveness and overhead of ProvTalk and DominoCatcher [106].

requires an insignificant additional amount of storage (around 0.2 megabyte) compared with ProvCloud due to the small number of API calls related to SFC-level. Fig. 42d shows that, on average, ProvNFV consumes around 0.5% higher CPU resources than DominoCatcher and other implemented levels of ProvTalk due to capturing the dependencies across all levels. The CPU consumption of ProvSFC is close to DominoCatcher due to the lower number of operations at SFC-level. In conclusion, we can see that the low storage and CPU consumption of ProvSFC in addition to its effectiveness (Fig. 42a, 42b) demonstrate the benefit of our solution for virtual environments with only the cloud and SFC levels. Although ProvNFV has a slightly higher storage and CPU cost, it leads to significant reduction (twice that of DominoCatcher).

4.9.4 Correctness

To answer RQ4, we evaluate how completely and soundly ProvTalk can capture and process changes in the NFV stack.

Completeness. Our analysis shows that all operations directed through NFV management interfaces are passed as API calls to the endpoint services to be applied to the NFV stack. Deploying ProvTalk as a middleware attached to all those services ensures the completeness property as it can capture all management API calls (i.e., 100% coverage). Table 12 shows the number of unique types of management API calls (according to Tacker-OpenStack documentation [3]) that are issued to most commonly used services. Moreover, we conduct an exhaustive study of all database tables in the NFV platform and devise an entity-relationship (ER) model to ensure all cross-level relationships are captured by constructed provenance graphs.

Table 12: The number of types of management API calls.

Services	**Tacker**	**Nova**	**Neutron**	**Glance**	**Swift**	**Heat**
Unique API calls	73	313	251	31	16	58
Coverage (%)	100	100	100	100	100	100

Soundness. ProvTalk stores the interception time of API calls at all services to preserve the temporal order of events during backtracking on the provenance graph. Moreover, our pruning scheme preserves the soundness property by leveraging the cross-level dependencies as follows. At the same level as where the incident is detected, ProvTalk prunes only the nodes identified to be irrelevant according to the defined pruning policies (which is consistent with [106, 92]). Our multi-level pruning leverages cross-level dependencies to only prune the nodes at other levels that correspond to the irrelevant nodes identified by the existing techniques. Also, our aggregation schemes ensure the soundness property by preserving all the relationships between the aggregated nodes and the rest of the provenance graph. In other words, all edges pointing to/from each group of aggregated nodes will point to/from their representative compound node, and the compound nodes can be expanded to show the aggregated nodes in the original form.

4.9.5 User Studies

To answer RQ5, we conducted two user studies[1] based on standard practices [14] in which the participants have to identify the root cause of an incident using ProvTalk. In our first study (static outputs), participants are provided with the outputs of ProvTalk that we obtained in advance. In our second study (live interaction), participants could directly interact with ProvTalk, trigger each module, and/or analyse the output of customized queries (e.g., nodes related to operations requested by non-admin users).

Table 13: Statistics of participants. PG means provenance-based analysis. (A), (L) and (N) mean advanced, little and no knowledge, respectively. Numbers of participants are shown in the first row of the tables related to each study.

1st (Static outputs)	**Industry (14)**					**Academia (7)**	
Background (NFV-PG)	A-A	A-L	A-N	L-L	L-N	A-L	L-L
Participants (%)	14	19	5	19	9	5	29
Scores	3.89	4.19	4	4.1	4.38	3.92	4.26

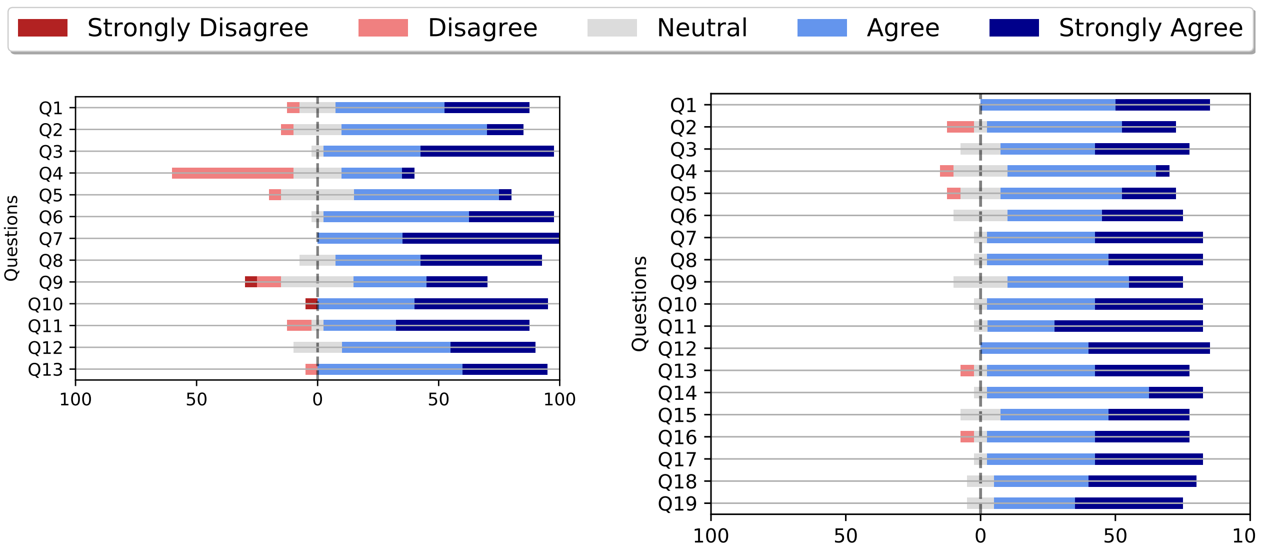
2nd (Live interaction)	**Industry (6)**			**Academia (11)**			
Background (NFV-PG)	A-A	A-L	A-N	A-L	L-L	L-N	N-N
Participants (%)	12	17	6	35	12	12	6
Scores	4.78	3.91	4.36	4.14	4.44	4.28	4.05

Participants. To conduct both studies, we recruited participants from a telecommunication industrial organization and graduate students working in cybersecurity from our university. Table 13 shows the statistics of participants in each study and the average score for all provided statements.

In the beginning of both studies, we provided a brief review of an attack story (our motivating example in Section 4.2.2), and asked the participants to express their level of agreement with our provided statements. Our web-based platform showed the outputs generated by ProvTalk in split-screen together with the statements. Table 14 shows the list of statements common between our two studies. Table 15 shows the additional statements specific to our second study (live interaction). Participants could express their agreement level by choosing either *Strongly agree*, *Agree*, *Neutral*, *Disagree* or *Strongly disagree*. To quantify the results, we calculated the average scores

[1]Those studies have been approved by Research Ethics/Office of Research of our university.

by assigning an integer between one and five to each option (five represents *Strongly agree* and one represents *Strongly disagree*).



(a) Based on the first study. (b) Based on the second study.

Figure 43: Participants' agreement with the statements.

Results. According to Table 13, ProvTalk achieves scores above 3.8 among all groups. Fig. 43 shows the distribution of participants' agreement with statements of our studies. Based on our results, pinpointing the root cause is challenging for most participants using the multi-level provenance graph (Q1-3). The results from Q4 and Q5 demonstrate the advantage of our multi-level pruning over the existing schemes in making the provenance graph easier to understand. The results (Q6-9) affirm the analysis becomes easier with our aggregation schemes by decreasing the size of the provenance graph and assigning expressive labels to compound nodes (i.e., adding semantics). Additionally, most participants find the expansion of compound nodes helpful for detailed analyses (Q10). The generated textual description facilitates the analyses for most participants (Q11-12), and they can easily associate the descriptions with their corresponding parts of the provenance graph for detailed analyses (Q13). We show the average quantified score for the aforementioned statements in Table 14.

Added benefit of live interaction. There is a slight difference between agreement levels in the two studies, with a more significant gap in some cases, especially Q4 and Q9. We believe that the

Table 14: Statements common in our two studies. To quantify the results, we convert participants' agreement level to scores between one and five (score five represents *Strongly agree*). ScoreS and ScoreL represent the scores of our first and second studies, respectively.

Module	**Statement**	**Code**	**ScoreS**	**ScoreL**
Multi-level Provenance	Given the incident and the provenance graph at each level, it is almost impossible to find the root cause.	Q1	4.2	4.41
	Given the incident alert and the connections between levels, I could find the path from the incident to the root cause.	Q2	3.85	3.94
	It is time-consuming to recognize the root cause among all graph nodes.	Q3	4.4	4.24
Pruning	One-level pruning made it easier to identify the attack-related graph nodes.	Q4	2.85	3.71
	Multi-level pruning made it easier to identify the attack-related graph nodes (w.r.t one-level pruning).	Q5	3.65	3.94
Aggregation	Cross-level aggregation made understanding the relationship between nodes at different levels easier.	Q6	4.3	4.11
	The graph seems less complex after cross-level aggregation.	Q7	4.65	4.41
	The labels inside the compound nodes are helpful in understanding the provenance graph.	Q8	4.35	4.35
	It was easier to find the root cause after aggregating and labeling administrative behavior-related nodes.	Q9	3.6	4
	Expanding the compound nodes can provide useful details in an on-demand basis.	Q10	4.4	4.41
Rule-based Translation	Seeing this generated textual description would have made identifying the root cause much easier.	Q11	4.3	4.58
	The generated text is similar to what was described about the attack story in the beginning.	Q12	4.15	4.52
	It is easy to map the summary with the provenance graph.	Q13	4.25	4.23

improved results in our second study (live interaction) is due to the added capabilities of participants to directly play with all modules or make customized queries (as shown by Q14-15). Additionally, interactive features such as zooming and node dragging render the effectiveness of ProvTalk more visible to participants (as shown by Q16-Q19).

Table 15: Statements specific to our second study. Scores are between one and five (score five represents *Strongly agree*).

Statement	**Code**	**Score**
Live interaction with ProvTalk enabled searching for certain relationships between nodes (e.g., those satisfying given properties), which made the analysis easier.	Q14	4.17
Live interaction with ProvTalk enabled highlighting certain relationships between nodes (e.g., those satisfying given properties), which made the analysis easier.	Q15	4.17
Hovering over nodes shows all of their properties, which enables easier access to more information.	Q16	4.23
Zooming feature enhanced the visibility of different nodes.	Q17	4.41
Node dragging feature made understanding the relationships between nodes easier.	Q18	4.35
Aggregation/compression of any node of my choice made the analysis easier.	Q19	4.37

Based on the results of our studies, we conclude that ProvTalk can effectively facilitate the analysis for both expert and non-expert users.

4.10 Discussion

We discuss limitations and future directions of ProvTalk.

Integration with system-level solutions. ProvTalk can potentially be integrated with system-level provenance solutions to provide mutual benefits (e.g., ProvTalk can guide system-level solutions to focus on specific resources, whereas the latter can corroborate the findings of ProvTalk with low-level details). Moreover, we envision that the system calls captured within virtual resources (e.g., using SPADE system [38]) can be integrated with the provenance graph generated by ProvTalk in a manner consistent with previous works (e.g., [44, 74]).

Other platforms. While we focus on OpenStack-Tacker platform in this work, our approach can be extended to other NFV platforms with an initial effort of adapting to their management operations. For instance, in AWS cloud, we could map the NFV-level of our provenance model to AWS CloudFormation services [102]. Our model can also be mapped to container-based platforms such as Kubernetes-Tacker [89]. Additionally, as we show in Section 4.9.4, the only engineering effort required for collecting the information of our interest is to plug in API interceptors suitable for the studied platforms (e.g., AWS Lambda in AWS cloud [103]).

More complex modeling approaches. Since ProvTalk focuses on assisting, instead of replacing human analysts, we are less concerned with the accuracy of our aggregation approach. Nevertheless, in our future work, we will investigate the effect of using different learning and embedding techniques (e.g., [47, 77]) on the reduction power. Moreover, we plan to extend ProvTalk to automatically detect anomalous behavior [118, 42] and translate it into a human-readable text [30]. Additionally, we will apply adversarial machine learning to study potential attacks (e.g., adversarial event sequences) against our solution.

Limitations around limited coverage and applicability. ProvTalk does not cover system-level events inside individual virtual resources, and it can potentially be integrated with system-level provenance solutions in order to address the scalability issue of the latter in a complex NFV system. To maintain the applicability of ProvTalk, analysts will need to periodically update the models trained by CDM and ABM modules. Finally, ProvTalk is designed to facilitate the investigation of management operations by human analysts instead of replacing them.

4.11 Related Work

NFV incident investigation. Existing role/permission-based techniques (e.g., [11]) focus on incident prevention, and cannot be applied to investigate the attacks bypassing such techniques [45]. Several incident investigation solutions have been proposed for NFV platforms (e.g., [100, 53]) to enable locating malfunctioning components through alert correlation techniques. Chain-Guard [33] and SFC-Checker [111] verify the correct forwarding behavior of service function chains. vSFC [126] enables identifying a wide range of security threats (e.g., packet injection attacks). Unlike ProvTalk, these solutions do not directly identify the root cause operations.

Provenance-based solutions. Provenance-based security analysis has been extensively studied in the literature [49, 94, 92, 91, 43, 44, 42, 72]. The authors in [92, 40, 91] improve the capture mechanism by building the provenance graphs based on the information captured by Linux Security Module hooks. LPM [16] leverages data provenance to ensure authenticated system communications. CamQuery [92] increases the efficiency of provenance analyses through tracing both userspace and in-kernel executions. As a management-level solution, ProvTalk may work in tandem and complement those system-level techniques.

Past frameworks for layered provenance (e.g., [62, 74, 44]) integrate application logs into the OS-level provenance of hosts to enable more accurate analysis by removing irrelevant dependencies. The authors in [43] and [118] leverage OS-level provenance to triage alerts and detect exploited instances of a program instead of root cause analysis. There exist network provenance-based techniques [23, 22] focusing on network traffic and reference packet events instead of management operations initiated by users (which is the focus of ProvTalk). ProvThings [117] proposes a provenance-based approach for auditing the IoT applications across different devices. In SDN environments, FORENGUARD [115] provides flow-level forensics and ProvSDN [112] monitors the access to sensitive data for unprivileged applications. In [121], the authors identify the absence of events in distributed systems. In [42], the authors produce a behavioral model of distributed applications to identify anomalous events in a cluster. Bates et al. [15] propose a provenance-based access control mechanism ensuring cloud storage security. The authors in [81] propose a tenant-aware solution to enhance OpenStack access control mechanism. DominoCatcher [106] tracks cloud management operations in single-level provenance graphs. Our experiments show that our approach can reduce the size of cloud-level provenance graphs twice as much as DominoCatcher does (see Section 4.9.3.3 for detailed results). Moreover, unlike our work, none of these solutions can support tracking information flow to/from different visualizations of the same resources that we face in NFV. Finally, summarization solutions (e.g., [124]) abstract user behaviors using audit logs. However, such abstraction is already intrinsically provided by NFV stack. ProvTalk leverages the distinct visualizations of resources that already exist at different levels of the NFV stack to infer the semantics of lower-level events and facilitate the analyses.

4.12 Conclusion

In this chapter, we presented ProvTalk, the first multi-level provenance solution for NFV. ProvTalk leveraged data provenance concept to find the management operations leading to attacks in NFV platforms and provided efficient pruning, aggregation and translation mechanisms for users to pinpoint the root cause of security incidents. We integrated ProvTalk to Tacker-OpenStack and demonstrated the efficacy of our approach based on real attack scenarios. Moreover, based on our experiments on performance and storage cost, our system substantially reduces the size of the provenance graph with insignificant runtime and storage overhead. Finally, our user studies results show that our approach remarkably facilitates the identification of root cause of security incidents.

Chapter 5

VinciDecoder: Automatically Interpreting Provenance Graphs into Textual Forensic Reports with Application to OpenStack

5.1 Introduction

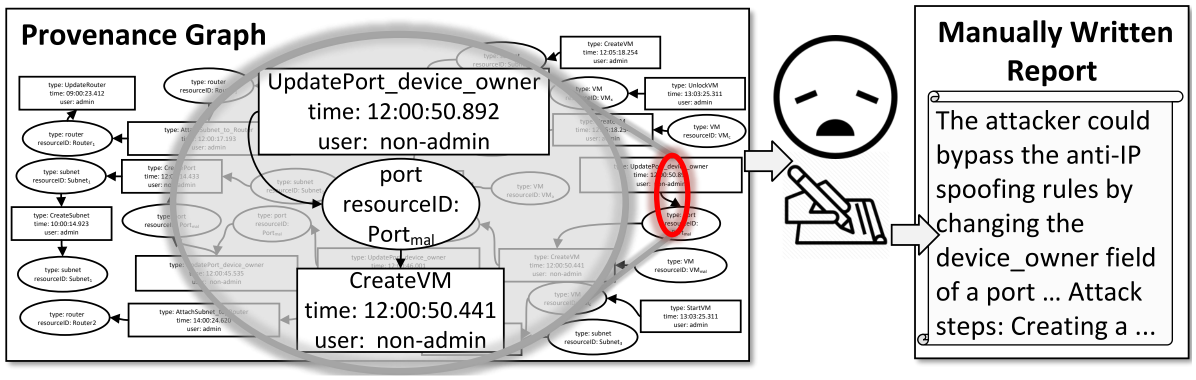
With the recent worldwide surge in adopting cloud computing, there is an increasing need for explaining the root cause of security incidents in large scale cloud infrastructures [1]. Sharing detailed forensic reports about such root causes and attack techniques can raise cybersecurity awareness, and improve threat detection and attack prevention techniques [48]. However, most existing provenance-based solutions (e.g., [117, 106, 91]) would face a critical challenge in such a context, i.e., it would be impractical to rely on human analysts to interpret the large and complex provenance graphs produced by such solutions for a large cloud with tens of thousands of inter-connected virtual resources [70].

There exist rule-based approaches (e.g., [109]) for generating textual summaries of provenance graphs. However, only relying on a set of specified rules [109] would not be sufficient, as the unpredictable nature of security incidents [122] will necessitate to constantly develop new rules, which may be costly especially for large clouds. We will further illustrate such limitations through the following example.

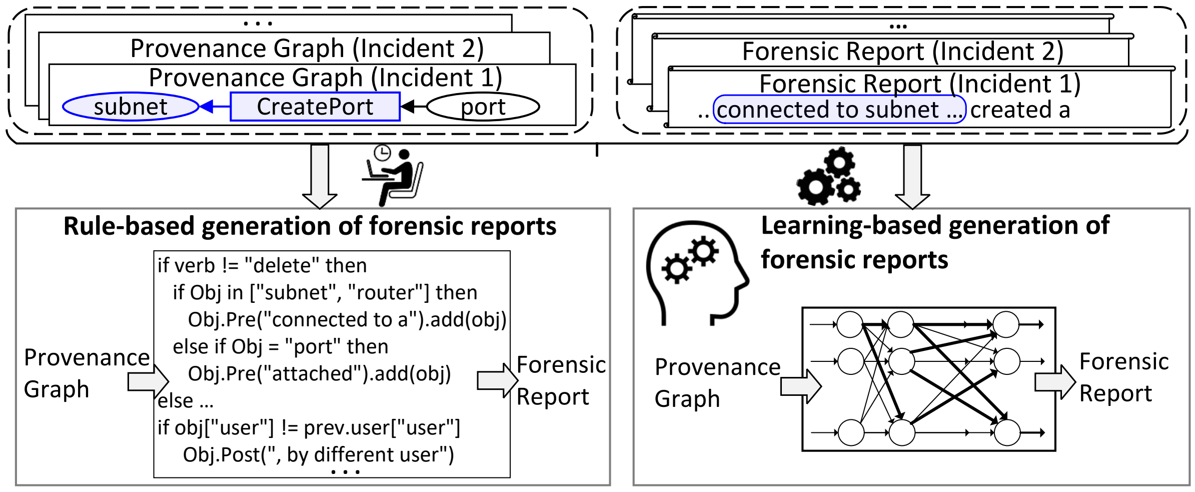
Motivating example. Fig. 44(a) depicts a provenance graph (left), and an analyst manually performing the task of creating a human-readable report (right) based on the provenance graph. Specifically, upon receiving an alert about the leakage of network traffic, the analyst begins investigating the suspicious paths of the provenance graph (left) generated by existing tools (e.g., [118]) to manually report the root cause as shown in Fig. 44a (right) (the exploit of a vulnerability [12] by updating the *device_owner* field of a port attached to a created VM). However, such a task can be challenging to an analyst, especially as a real world cloud provenance graph may have tens of thousands of nodes and edges [106].

- **Key ideas.** Fig. 44(b) shows the two main approaches adopted by our solution, namely *VinciDecoder* [108], for automatically interpreting provenance graphs into forensic reports. First, our rule-based approach generates customized forensic reports based on lexicons and grammar rules as illustrated in Fig. 44(b) (bottom left). Such rules are specified by the analyst according to his/her criteria (e.g., domain-knowledge) and understanding of the existing paired provenance graphs and forensic reports for similar types of future attacks. Second, for use cases where such criteria are too dynamic (e.g., new types of attacks) for a rule-based approach to handle, we also propose a learning-based approach (bottom right) which automatically learns the correspondence between pairs of provenance graphs and forensic reports using Neural Machine Translation (NMT). Specifically, similar to words (e.g., verbs and object) of a sentence, there is a dependency between nodes (e.g., operations and their affected resources) in a provenance graph, which inspires us to train a translation model by applying NMT to provenance graphs (source language) paired with human-readable reports (target language), and automatically translate future provenance graphs into a natural language interpretation using the trained model.

- **Challenges.** Although our vision for adopting NMT seems plausible, realizing VinciDecoder



(a) Challenges of interpreting provenance graphs: an excerpt of the provenance graph (left); an analyst manually creating a report based on the provenance graph (right).



Provenance Graph (Incident 2)
Provenance Graph (Incident 1)
subnet
CreatePort
port
Forensic Report (Incident 2)
Forensic Report (Incident 1)
.. connected to subnet ... created a
Rule-based generation of forensic reports
Provenance Graph
Forensic Report
Learning-based generation of forensic reports
Provenance Graph
Forensic Report

```
if verb != "delete" then
   if Obj in ["subnet", "router"] then
      Obj.Pre("connected to a").add(obj)
   else if Obj = "port" then
      Obj.Pre("attached").add(obj)
else ...
if obj["user"] != prev.user["user"]
      Obj.Post(", by different user")
      ...
```

(b) Our main idea: provenance graphs of several incidents (top left); existing forensic reports (top right); automatic generation of forensic reports (bottom left and right).

Figure 44: Motivating example.

requires addressing the following two main challenges. First, NMT is typically applied to textual data, whereas provenance graphs are usually stored as nodes and edges. To address this, VinciDecoder converts paths of provenance graphs into primitive sentences of node properties (detailed in 5.3.2). Second, it is challenging to generate high quality reports with a limited number of paired provenance graphs and reports for training. To address this, VinciDecoder leverages tens of thousands of CVE entries and their corresponding provenance graphs to train NMT (detailed in 5.4.2).

In summary, our main contributions are as follows:

– To the best of our knowledge, VinciDecoder is the first solution for generating forensic reports based on provenance analysis results using both rule-based and learning-based techniques. By reducing the reliance on human analysts to interpret and document large and complex provenance graphs, our approach can avoid the limitations, human error, and delay that are natural to such manual efforts, and thus improve the practicality of provenance analysis in large-scale cloud environments, enable automated documentation of root causes for security incidents, and allow for more timely incident-response.

– To automatically generate reports using NMT, we design several mechanisms as follows. VinciDecoder first converts provenance graph paths into primitive sentences representing properties of nodes, and removes instance-specific information to avoid mis-translation; it then learns a translation model based on the paired primitive sentences and reports; finally, when given target paths, VinciDecoder applies the learned model to the primitive sentences representing the paths to generate the forensic report. Optionally, our rule-based approach can form forensic reports by linking the node properties extracted from the target path based on pre-specified rules.

– We implement VinciDecoder on an OpenStack-based cloud testbed, and validate its effectiveness based on real-world security incidents. Our experiments and user study show that VinciDecoder generates high-quality results (e.g., up to 0.68 BLEU score for precision) with sufficient readability for human analysts (e.g., 92% of our participants agree that understanding the attack steps is much easier using VinciDecoder's report than using provenance graphs).

The rest of this chapter is organized as follows: Section 5.2 provides some background on data provenance and NMT. Section 5.3 details our methodology. Section 5.4 describes our implementation and presents the evaluation results. We discuss different aspects of our work and the related work in Section 5.5 and Section 5.6, respectively. We conclude the chapter in Section 5.7.

5.2 Preliminaries

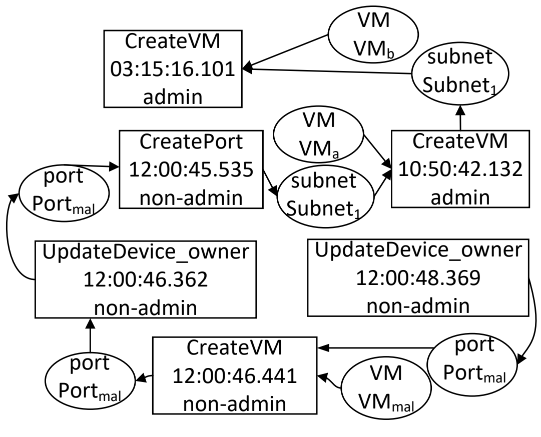
This section provides a background on data provenance, NMT and our assumptions.

5.2.1 Provenance Graph

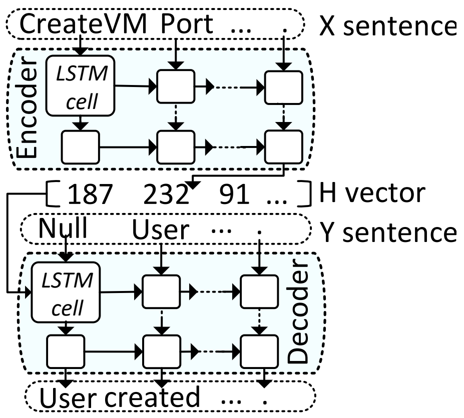
As a powerful technique to capture the dependencies between data objects (e.g., virtual resources or operating system files) and events (e.g., management operations or system calls) in a graph representation, data provenance has been applied to clouds. We show an example of a cloud management-level provenance graph [106] in Fig. 45(a) consisting of two types of nodes: *entities* (shown as ovals) and *activities* (shown as boxes), where entities represent virtual resources (e.g., a virtual port $Port_{mal}$), and activities represent cloud management operations (e.g., an operation *CreateVM*). Each node stores several properties such as the type of the operations/resources and the user who triggers the operations. Edges indicate the dependency between an operation and its affected resources. For example, in Fig. 45(a), the edge from *CreateVM* to $Port_{mal}$ shows that this operation attaches $Port_{mal}$ to the created VM VM_{mal}.

5.2.2 Neural Machine Translation

Neural Machine Translation (NMT) [105] builds a conditional probability model, $P(Y|X)$, such that the likelihood of a target sentence Y given a source sentence X is maximized [46]. As Fig. 45(b) shows, NMT usually consists of an encoder and a decoder, which typically utilize recurrent neural networks (RNN) such as a Long Short-Term Memory (LSTM) [47]. To initialize the training, LSTM cells are assigned with random *weights*, and the encoder captures the semantics of X by encoding it into a fixed-length vector H. Then, the decoder generates the target sentence given the computed vector H. NMT computes the deviation of the generated sentence from the reference sentence Y and improves the model by optimizing the assigned weights based on other pairs of sentences. In Section 5.3.4 and 5.3.5, we detail how VinciDecoder leverages this mechanism to generate forensic reports.



(a)



(b)

Figure 45: An excerpt of a cloud management-level provenance graph (a); an example of NMT Encoder-Decoder model (b).

5.2.3 Assumptions

We assume the accuracy of provenance analysis results provided by existing tools (e.g., [118]), such as suspicious paths capturing the attack steps or malicious behavior. We also assume the correctness and completeness of provenance-based root cause analysis solutions (e.g., [118, 42]) in identifying suspicious paths capturing the attack steps. We assume that the provenance construction tool is not compromised. Finally, similar to most other learning-based data-to-text techniques (e.g., [95]), we assume the availability of a sufficient amount of training data (i.e., paired forensic reports and suspicious paths) for training our model[1].

5.3 VinciDecoder

In this section, we provide an overview of VinciDecoder, detail its different modules, and describe the interaction between them.

5.3.1 Overview

Fig. 46 shows an overview of VinciDecoder, which consists of two main phases: learning phase and automatic report generation phase. In the learning phase, VinciDecoder collects paired suspicious

[1]In Section 5.4.2, we discuss how we obtain more pairs of reports and paths for training.

paths and reports for training, and then transforms suspicious paths into primitive sentences in our intermediary language (Section 5.3.2), which represents the properties of a node as a compound word, and removes the instance-specific information (Section 5.3.3). Next, it applies NMT to train a translation model profiling the correspondence between the obtained sentences and their forensic reports (Section 5.3.4). In the automatic report generation phase, VinciDecoder applies the trained translation model to generate forensic reports based on the suspicious paths of the provenance graph associated with the newly detected incident (Section 5.3.5). Optionally, VinciDecoder can generate reports using our rule-based mechanism (Section 5.3.5).

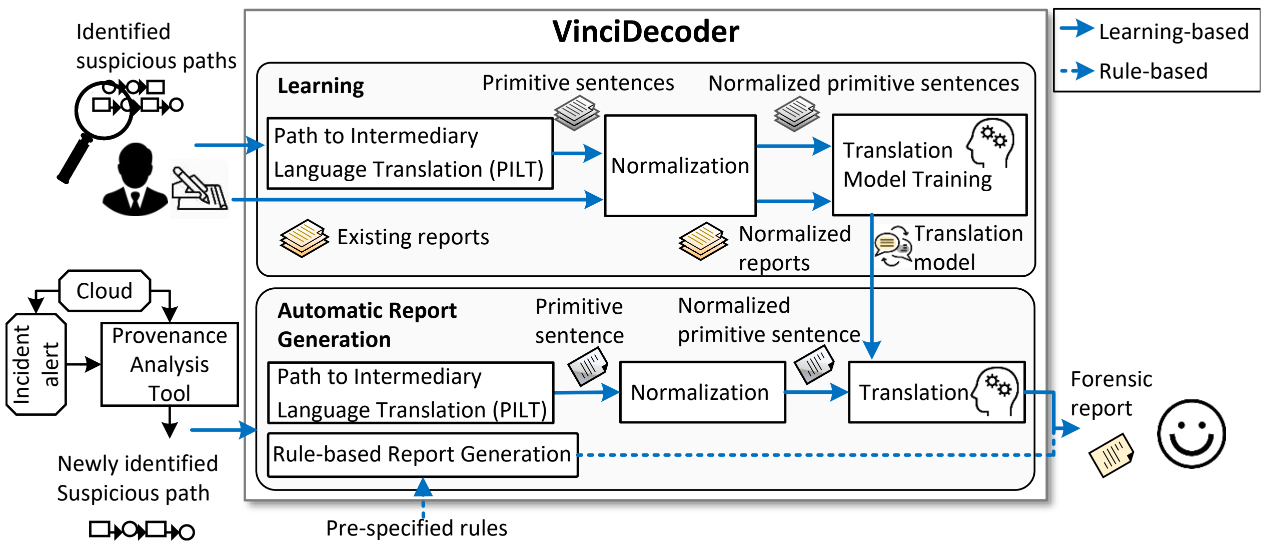

VinciDecoder
Learning-based
Rule-based
Identified suspicious paths
Learning
Primitive sentences
Normalized primitive sentences
Path to Intermediary Language Translation (PILT)
Normalization
Translation Model Training
Existing reports
Normalized reports
Translation model
Cloud
Incident alert
Provenance Analysis Tool
Automatic Report Generation
Primitive sentence
Normalized primitive sentence
Path to Intermediary Language Translation (PILT)
Normalization
Translation
Rule-based Report Generation
Forensic report
Newly identified Suspicious path
Pre-specified rules

Figure 46: Overview of VinciDecoder.

5.3.2 Path to Intermediary Language Translation (PILT)

NMT can be applied to textual sentences, which renders its application to provenance graphs challenging. To address this, the PILT module converts each suspicious path into a primitive sentence by querying the database to sequentially scan the nodes, extract their properties, and record them as one compound word of the sentence (see Fig. 46). Algorithm 2 details the mechanism of PILT as follows: PILT extracts the properties *type* and *user* from operation nodes and appends them to the created primitive sentence (line 3-5). Moreover, it calculates the *elapsed time* between the timestamp properties stored at two consecutive operation nodes (line 6-7) and appends the calculated value with a proper post-fix (e.g., “-millisecond", “-hours", etc.) to the sentence (line 8-9). The elapsed time may be interesting for reporting the incidents where the attacker attempts to issue a large number of operations in a short period of time, e.g., to launch race condition or DoS attacks. PILT also records the *type* and the identifier of resources (line 10-13). In the next section, we detail how obtained sentences are modified and leveraged to train the translation model.

Algorithm 2 Path to Intermediary Language Translation

```
Input:   path ← Suspicious path identified by the provenance analysis tool
Output:  SenRepresentingPath
1.  foreach node ∈ path do
2.      if isOperation(node) then %Appending the operation properties to sentence
3.          OperationType ← node["data"]["OperationType"]
4.          User ← node["data"]["user"]
5.          SenRepresentingPath.append("type:" + OperationType + "user:" + User)
            %Appending the approximate elapsed time between operations to sentence
6.          if isNotFirstNode(node) then
7.              ElapsedTime ← ThisOperation – PreviousOperationTime
8.              ElapsedApprox ← ElapsedTimeApproximator (ElapsedTime)
9.              SenRepresentingPath.append(ElapsedApprox)
10.     else if  isResource(node) then%Appending resource properties to sentence
11.         ResourceType ← node["data"]["ResourceType"]
12.         ResourceID ← node["data"]["ID"]
13.         SenRepresentingPath.append("type:" + ResourceType + "ID:" + ResourceID)
14. return SenRepresentingPath
```

Example 1. Fig. 47 shows the translation of a path (left) into a primitive sentence (right) in our intermediary language. As we can see, the properties of each node (e.g., the node representing *CreatePort* operation) are represented by a word (e.g., *“type:CreatePort,user:non-admin"*) in the obtained sentence.

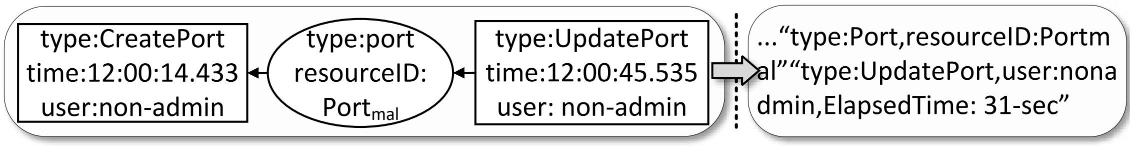


Figure 47: Simplified example path (left) translated into a primitive sentence (right).

5.3.3 Normalization

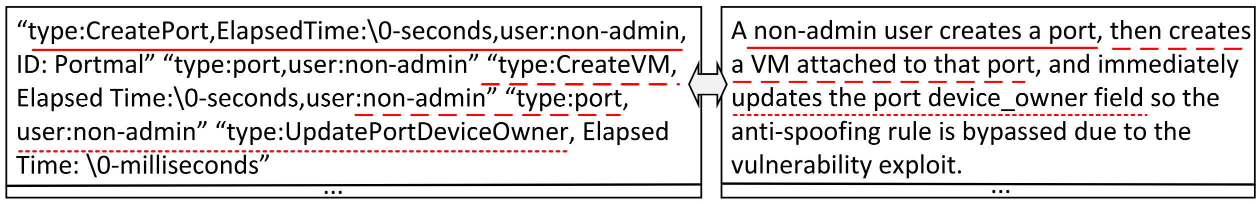
To allow NMT to focus on generic words in the primitive sentences instead of application-specific ones (which may lead to mis-translation), VinciDecoder needs to remove instance-specific information from the dataset before feeding it to NMT. Specifically, the forensic reports and their corresponding primitive sentences used for training may contain values (e.g., the name/ID of resources) that are related to semantics of the specific scenarios (which NMT is not aware of). Retaining such values is known to reduce the quality of the reports generated by the trained neural translation model [98]. Therefore, VinciDecoder identifies and replaces all instance-specific values (e.g., the number preceding the string "-milliseconds"[1]) with a placeholder (i.e., \0), and the name of the applications or software platforms with the word "*platform*" based on our specified rules.

5.3.4 Translation Model Training

This module builds a translation model to profile the correspondence between the existing forensic reports and their associated suspicious paths. To this end, we leverage NMT [50], as it automatically captures the *context* of words and nodes (i.e., the dependencies between words in a report and nodes in a path) using embeddings. By applying NMT, VinciDecoder first projects words of a report and the words of the obtained primitive sentences (i.e., properties of nodes) into a high-dimensional numerical vector space such that words/nodes with similar contexts have closer vector representations. Next, VinciDecoder builds a translation model based on the derived vectors that optimally maps each provided forensic report to its paired primitive sentence.

Example 2. Fig. 48 shows an excerpt of the training dataset composed of the primitive sentences obtained from the suspicious paths (left) and their corresponding manually created reports (right). The semantically related information on each side are illustrated with the same type of lines.

[1]Despite removing the numbers, the range of the elapsed time (e.g., milliseconds vs. hours) retains useful information about the incidents.


“type:CreatePort,ElapsedTime:\0-seconds,user:non-admin, ID: Portmal” “type:port,user:non-admin” “type:CreateVM, Elapsed Time:\0-seconds,user:non-admin” “type:port, user:non-admin” “type:UpdatePortDeviceOwner, Elapsed Time: \0-milliseconds”
...

A non-admin user creates a port, then creates a VM attached to that port, and immediately updates the port device_owner field so the anti-spoofing rule is bypassed due to the vulnerability exploit.
...

Figure 48: Example paths in our intermediary language (left) and their corresponding manually written reports (right). Semantically related information are highlighted by the same type of lines.

5.3.5 Automatic Report Generation

Once a new security incident is detected, VinciDecoder automatically generates forensic reports based on the suspicious path identified by existing tools (e.g., [118, 42]) using our learning-based and rule-based techniques.

Learning-based report generation. After building the translation model in the learning phase, VinciDecoder can be applied to generate forensic reports based on the detected suspicious path. Specifically, VinciDecoder converts the suspicious path into a primitive sentence in our intermediary language, and removes the instance-specific information by following the same techniques as mentioned in Section 5.3.2 and Section 5.3.3. Next, it applies the translation model to each normalized primitive sentence to automatically generate the corresponding forensic report. To improve the quality of generated reports, VinciDecoder also allows the analyst to conduct post-editing [54] by identifying the instance-specific information using the primitive sentences and adding them to the reports.

Rule-based report generation. To ensure the applicability of our approach when there is a lack of a sufficient number of reports for training, VinciDecoder is also equipped with a rule-based mechanism, which enables translation without training data. Specifically, VinciDecoder sequentially scans nodes on each path, and extracts the following properties stored at each node: the *type* and *ID* of resources/operations, the *user* triggering an operation, and the *elapsed time* between the timestamp values stored in two consecutive operation nodes. Then, it creates an ordered list, where each item represents the properties of a node. Next, VinciDecoder generates sentences based on our specified rules by sequentially linking the items such that the extracted user, resource, operation and elapsed time are included as the subject, object, verb and propositional phrase in generated sentences, respectively (detailed in Appendix). Finally, an introductory and a concluding sentence are generated to describe an overview of the incident (e.g., describing the time of the detection).

Algorithm 3 shows our rule-based mechanism generating reports based on the cloud management-level provenance graphs (e.g., the provenance graph in Fig. 44). To generate fluent sentences, we specify rules for indicating different subjects (line 2-5). We add resources extracted from the names of operations (e.g., a *VM* in *CreateVM*) through the template *a $resource_type named $main_resource_name* (line 7-9). We specify various rules (line 11-20) for describing other affected resources connected to an operation node. We also specify rules to record other information such as the elapsed time between operations (line 21-26). Through such rules specifically designed for each type of operations, resources, and users, VinciDecoder generates reports when there is an insufficient amount of training data for generating high quality reports.

Example 3. Fig. 49 shows the report related to the incident in our motivating example (Section 5.1). The report starts with explaining the number of operations in the suspicious path, continues with describing the attack steps, and concludes with indicating the ID of nodes in the suspicious paths.

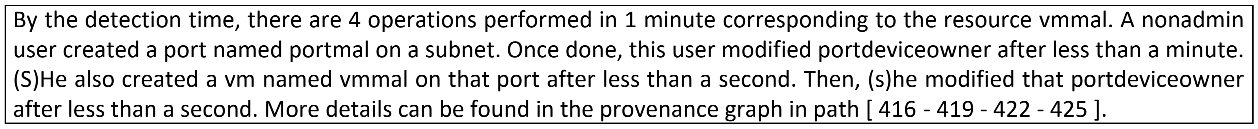

By the detection time, there are 4 operations performed in 1 minute corresponding to the resource vmmal. A nonadmin user created a port named portmal on a subnet. Once done, this user modified portdeviceowner after less than a minute. (S)He also created a vm named vmmal on that port after less than a second. Then, (s)he modified that portdeviceowner after less than a second. More details can be found in the provenance graph in path [416 - 419 - 422 - 425].

Figure 49: Automatically generated report on the incident discussed in our motivating example (Section 5.1).

5.4 Implementation and Evaluation

In this section, we detail the implementation of VinciDecoder and evaluate our solution.

Algorithm 3 Rule-based Report Generation

```
Input:    path ← Suspicious path identified by the provenance analysis tool
          Middle_Sentence_Subjects = ["Next, this user", "Later, he/she", "He/She also",
          "This user then", "Once done, he/she "]
Output:  Description
1.  foreach node ∈ path do
2.    if isFirstNode(node) or isNotEqualPreviousUser(node) then%User of the first operation
3.        Subj_main ← Admin_NonAdmin_Specifier(userID, adminID)
4.    else %Other users with prior words (e.g., "Once done, he/she")
5.        Subj_main ← random_choice(Middle_Sentence_Subjects)
6.    if isAnOperation(node) then
7.      Verb, MainObject ← OperationType.split(operation)
8.      MainObject ← MainObject.setDeterminer("a")
9.      MainObject ← addAfter("with the ID " + MainObject["id"])
10.     OtherAffectedResources ← EndOfOutgoingEdges(node)
11.     foreach SecondaryObject ∈ OtherAffectedResources: %Choosing prior words
12.        if verb != "delete" then
13.           SecondaryObject.addBefore("on").addAfter(resource["id"])
14.        else
15.           SecondObject.addBefore("from a").addAfter(resource["id"])
16.        if previousUser(resource) != operation["user"] then %Update by a different user
17.           SecondaryObject.addAfter(", previously affected by a different user,")
18.        if isNotFirstNode(node) then %Range of elapsed time between operations
19.           ElapsedTime ← TimeRangeDescriptor(ThisOperation – PreviousOperationTime
20.        if isAlertNode(node) then
21.           MainObject.addAfter(", which is associated to the alert.")
22.    sentence.setSubj(Subj_main).setVerb(Verb).setObj(MainObject) %Form sentence
23.    sentence.addAfter(SecondaryObject).addAfter(ElapsedApprox)
24.    if ThisOperation = PreviousOperation then %Emphasize the repetition
25.        sentence.addComponent("again")
26.        PathDescription.append(sentence)
27. return Description
```

5.4.1 Evaluation using Cloud Management-level Provenance Graphs

To evaluate VinciDecoder under different scenarios (e.g., various lengths of suspicious paths), we apply VinciDecoder to cloud management-level provenance graphs generated in our testbed cloud.

5.4.1.1 Implementation and Data Collection

We implement VinciDecoder in a cloud testbed based on OpenStack [83] (a popular open-source cloud platform). We note that only our PILT module (Section 5.3.2) and our rules (Appendix) are platform-specific, and the modular design of VinciDecoder makes it easily portable to other platforms or provenance models (e.g., OS-level provenance [49]). We export provenance graphs from Neo4j [80] into JSON format for processing. We leverage Open-Source Toolkit for Neural Machine Translation (ONMT) [50] (a popular tool for language translation). Similar to some other solutions (e.g., [118]), we choose the default options for embedding paths (i.e., 500 dimensional vector) as well as the batch size and the dropout rate (i.e., 64 and 0.3, respectively). We leverage the metrics in [104] to evaluate our approach.We run VinciDecoder on an Ubuntu 20.04 server equipped with 128 GB of RAM. We generate the provenance graphs through deploying and updating different types of virtual resources. Moreover, we enrich our training dataset by leveraging the rule-based mechanism (detailed in Section 5.3.5). To simulate reports authored based on various writing styles, we specify rules capturing the writing styles of our different authors.

Table 16: Statistics of our testbed datasets.

Training 2.5	**Dataset**	$D_{\text{tr-size1}}$	$D_{\text{tr-size2}}$	$D_{\text{tr-size3}}$	$D_{\text{tr-size4}}$
	# of paths	2000	4000	6000	8000
	$\mathbf{l_{min}}$	4	4	4	4
	Dataset	$D_{\text{tr-len1}}$	$D_{\text{tr-len2}}$	$D_{\text{tr-len3}}$	$D_{\text{tr-len4}}$
	# of paths	2000	2000	2000	2000
	$\mathbf{l_{min}}$	4	8	12	16
Testing	**Dataset**	D_{ts1}	D_{ts2}		
	# of paths	2000	2000		
	$\mathbf{l_{min}}$	4	8		

Table 16 shows the statistics of our datasets. To evaluate the effect of length and number of available samples (i.e., the suspicious paths) on the performance, we conduct our experiments based on two groups of training datasets: 1) varying the number of paths: four datasets ($D_{\text{tr-size1}}$ to $D_{\text{tr-size4}}$) each consisting of a different number of paths with the same minimum length; 2) varying the length of paths: four datasets ($D_{\text{tr-len1}}$ to $D_{\text{tr-len4}}$) consisting of the same number of paths with different specified minimum lengths. We randomly select 70% and 30% of the paths from each training dataset to build and validate (used by NMT to automatically tune the hyperparameters in training [46]) the models, respectively. Our training and testing datasets are selected from disjoint parts of the provenance graph, so we can evaluate the ability of VinciDecoder in handling unseen datasets. We also evaluate our approach based on two testing datasets with paths of different minimum lengths as shown in Table 16.

5.4.1.2 Effectiveness Evaluation

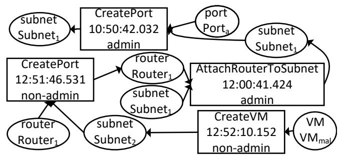
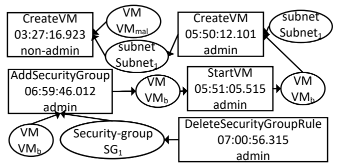
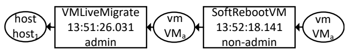
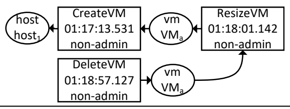
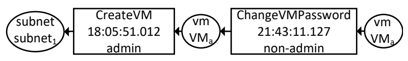
We reproduce in our testbed eight real-world incident scenarios that involve cloud management operations, and apply VinciDecoder to generate reports based on the captured provenance graphs. Table 17 shows those scenarios and corresponding incidents. Most of those scenarios are discussed in previous works (e.g., [110, 119, 64, 21, 107]) focusing on security verification. For all cases, our generated reports capture all operations that led to the incidents. Table 18 demonstrates the effectiveness of VinciDecoder based on five scenarios. We also showcased our result for the sixth scenario in Section 5.3.5. The other two scenarios (Table 17, seventh and eighth rows) involve fewer types of operations and are thus omitted due to space limitation.

Table 17: Attack scenarios used to evaluate the effectiveness of VinciDecoder.

Index	**Root cause**	**Incident**
1	Improper authorization [119]	Port Scanning
2	Failed update of security groups [107]	Data leakage
3	Soft-rebooting migrated VM [9]	Data corruption
4	Deleting resized VM [8]	Disk utilization
5	Incorrect role assignment [110]	Data leakage
6	Race condition in update port [107]	Data leakage
7	Wrong VLAN ID [21]	Data leakage
8	Excessive VM creation on a host [64]	Disk utilization

Example of verifying the captured information. Fig. 50(a) shows the automatically generated report explaining the operations (i.e., the creation of a rogue port on a router created by a different user) to exploit a vulnerability [7] that led to the attack on VM_a (Table 17, first row). VinciDecoder correctly details the steps described by the manually created report shown in Fig. 50(b).

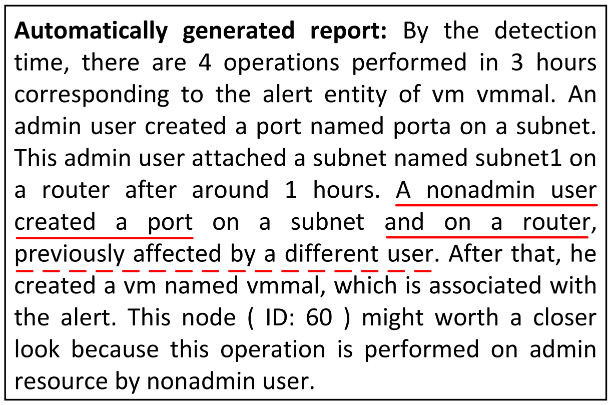
Table 18: Reports generated by VinciDecoder for five scenarios in Table 17. The sixth scenario is showcased in Section 5.3.5.

Index	Provenance graph path	Automatically generated report
1	 subnet Subnet1 CreatePort 10:50:42.032 admin port Porta subnet Subnet1 CreatePort 12:51:46.531 non-admin router Router1 AttachRouterToSubnet 12:00:41.424 admin subnet Subnet1 router Router1 subnet Subnet2 CreateVM 12:52:10.152 non-admin VM VMmal	An admin user created a port named porta on a subnet. This admin user attached that subnet to a router after around 1 hours. A nonadmin user created a port on that subnet and on that router, previously affected by a different user. After that, (s)he created a vm named vmmal, which is associated to the alert.
2	 CreateVM 03:27:16.923 non-admin VM VMmal subnet Subnet1 CreateVM 05:50:12.101 admin subnet Subnet1 AddSecurityGroup 06:59:46.012 admin VM VMb StartVM 05:51:05.515 admin VM VMb VM VMb Security-group SG1 DeleteSecurityGroupRule 07:00:56.315 admin	A nonadmin user created vm named vmmal on a subnet. An admin user created a vm named vmb on that subnet. Once done, (s)he started that vm vmb. Then attached a securitygroup named SG1 on that vm. The administrator deleted securitygrouprule from that SecurityGroup after around 1 minutes.
3	 host host1 VMLiveMigrate 13:51:26.031 admin vm VMa SoftRebootVM 13:52:18.141 non-admin vm VMa	An admin user livemigrated a vm named vma to a host. A nonadmin user softrebooted that vm, previously affected by a different user.
4	 host host1 CreateVM 01:17:13.531 non-admin vm VMa ResizeVM 01:18:01.142 non-admin DeleteVM 01:18:57.127 non-admin vm VMa	A nonadmin user created a vm named vma on a host. Later, (s)he resized that vm after less than a minutes. Next, (s)he deleted that vm after less than a minute.
5	 subnet subnet1 CreateVM 18:05:51.012 admin vm VMa ChangeVMPassword 21:43:11.127 non-admin vm VMa	An admin user created a vm, named vma on a subnet. A nonadmin user changedpassword a vm, previously affected by a different user after around 3 hours.

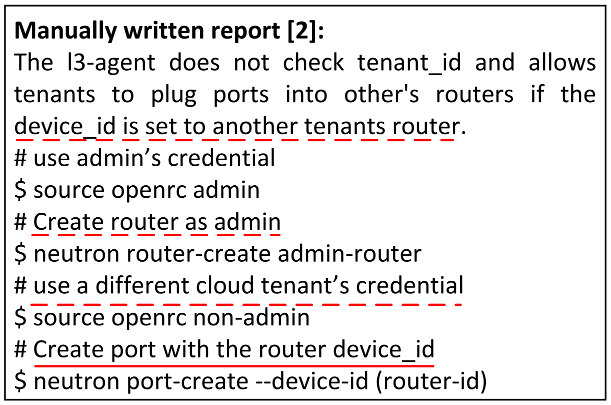
5.4.1.3 Performance Evaluation

We showcase the high quality of generated reports with different number and length of paths in training datasets based on well known translation metrics BLEU and ROUGE [104]. BLEU (*precision*) measures the fraction of the generated information that are relevant to the manually written reports and ROUGE (*recall*) indicates the fraction of information from the reference reports that are included in automatically generated reports. As VinciDecoder proposes the first learning-based provenance translation solution, we cannot directly compare our results to existing works, while we note that scores above 0.5 are generally known to reflect high quality translations [55].

Number of training samples. Fig. 51(a) shows that, in most cases, there is a minor variation in the evaluated performance as the number of the training samples increases. This can be explained by the possibility that our translation models trained by larger datasets may become more biased [61] due to the frequent appearances of similar patterns of cloud management operations. To further illustrate this effect, Fig. 52 compares an excerpt of a manually written report of a path with the


Automatically generated report: By the detection time, there are 4 operations performed in 3 hours corresponding to the alert entity of vm vmmal. An admin user created a port named porta on a subnet. This admin user attached a subnet named subnet1 on a router after around 1 hours. A nonadmin user created a port on a subnet and on a router, previously affected by a different user. After that, he created a vm named vmmal, which is associated with the alert. This node (ID: 60) might worth a closer look because this operation is performed on admin resource by nonadmin user.

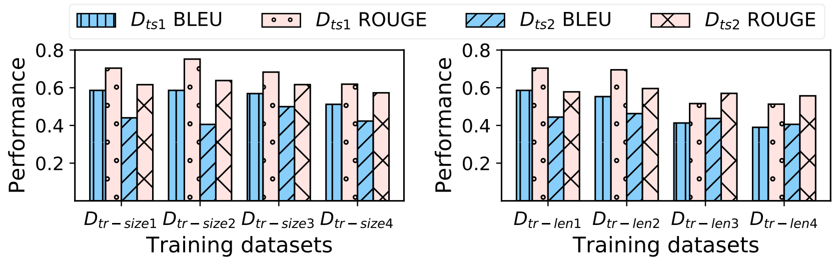
(a) Automatically generated report.


Manually written report [2]:
The l3-agent does not check tenant_id and allows tenants to plug ports into other's routers if the device_id is set to another tenants router.

```
# use admin's credential
$ source openrc admin
# Create router as admin
$ neutron router-create admin-router
# use a different cloud tenant's credential
$ source openrc non-admin
# Create port with the router device_id
$ neutron port-create --device-id (router-id)
```

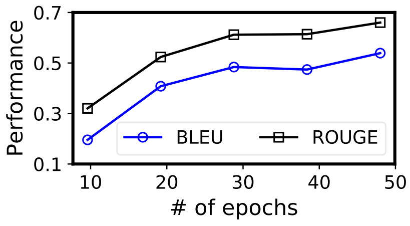
(b) Manually created report about exploiting the same vulnerability [7].

Figure 50: Verifying the information captured by our generated report. The semantically relevant information are highlighted with the same type of line.



(a) Effect of the number of training samples.

(b) Effect of the length of training samples.



(c) Effect of the number of epochs.

Figure 51: Evaluation with cloud management-level provenance graphs.

ones generated by VinciDecoder based on the four training datasets. As we can see, larger training datasets (e.g., $D_{\text{tr-size4}}$) cause more extra or missing information in the generated reports. We conclude that our approach remains useful even with a limited number of training samples.

Length of training samples. Fig. 51(b) shows that the performance decreases when the length of paths in the training dataset increases, which may be due to the degraded performance of NMT for longer sentences [27]. The reduction is more significant for D_{ts1} due to the difference between the length of paths in this testing dataset (with minimum four nodes) and that of paths in the training datasets, $D_{\text{tr-len3}}$ and $D_{\text{tr-len4}}$ (with minimum 12 and 16 nodes). The training datasets $D_{\text{tr-len1}}$ and $D_{\text{tr-len2}}$ (with paths of minimum four and eight nodes) cause a noticeably higher performance for D_{ts1} than for D_{ts2} due to the general positive impact of shorter paths of D_{ts1} on the performance and the similarity between training and testing datasets regarding the lengths of paths.

Path (converted to a sentence in the intermediary language): '"type:CreateServers,user:Admin, ElapsedTime:\0-seconds" "type:port" "type:UpdatePorts,user:Admin,ElapsedTime:\0-seconds"	
Manually written report: An admin user created a server attached to a port, named \0. He later updates that port after less than a minute.	
$D_{tr\text{-}size1}$: An Admin user created a server named \0 attached to that port. Once done, he updated a port named\0 [missing elapsed time].	**$D_{tr\text{-}size2}$:** An Admin user created a server named \0 attached to that port. [not using pronoun] The Admin user modified a port named\0, after less than a minute.
$D_{tr\text{-}size3}$: [missing create server] An Admin user updated a port named \0. ~~The administrator updated a port named \~~0 , after~~ around \0minute~~s.	
$D_{tr\text{-}size4}$: An Admin user ~~created a port named \0 connected to that subnet. Then created a port named\0 connected to that subnet. This Admin user created a port named \0 connected to that subnet after around minutes.~~ ~~Once done, he created a port named \0 connected to that subnet. He~~ created a server named \0 attached to that port after less than a minute. [not using pronoun] This Admin user modified a port named \0 after less than a minute.~~ Then updated a port named \~~0.	

Figure 52: Comparing reports generated by training datasets with different numbers of samples (irrelevant parts of the generated reports are crossed out).

Number of epochs. Fig. 51(c) shows that the performance is significantly improved with the number of epochs (i.e., the number of times NMT iterates through a training dataset). We also measure the perplexity (i.e., the extent a trained model could predict a newly provided data [24]) and the accuracy for different numbers of epochs and training datasets. Fig. 53(a) shows that the perplexity decreases to around 1.2 after 20 epochs, and Fig. 53(b) shows that the accuracy increases to around 97% after 40 epochs. We conclude that the perplexity and accuracy of VinciDecoder improve with the number of epochs, and reach almost constant values after training over maximum 40 epochs.

Out-of-vocabulary evaluation. Fig. 53(c) shows that, without normalization (Section 5.3.3), the size of the vocabulary significantly grows with the size of the dataset, which may subsequently reduce the performance. Furthermore, Fig. 53(d) shows that, on average, the proportion of unseen words in the testing dataset (i.e., words that do not exist in the training datasets, and thus may be translated incorrectly) is around 6% less after conducting normalization. This shows that our normalization technique effectively increases the applicability of the trained models for describing new provenance graphs in testing datasets. In summary, our results demonstrate the feasibility and

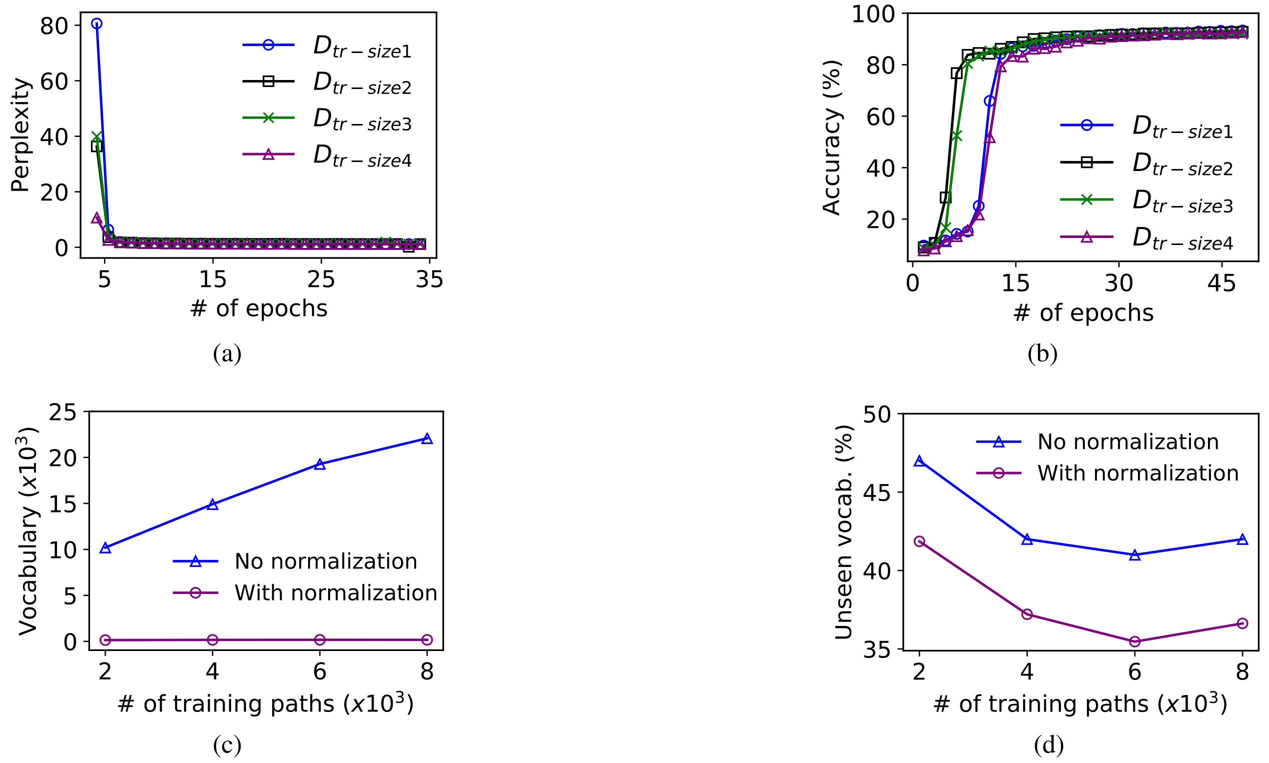


Figure 53: (a) Perplexity (the smaller is better) and (b) accuracy at different epochs; (c) the growth of vocabulary size, and (d) the proportion of unseen words.

quality of the produced reports for datasets with different number and length of paths.

5.4.2 Large Scale Experiments using CVE-based Provenance Graphs

As our evaluation in Section 5.4.1 is limited to the data collected from our testbed, to evaluate our approach based on more realistic and larger scale datasets, we apply VinciDecoder to CVE-based provenance graphs in this section.

Data Collection. The performance of NMT may be adversely affected by the scarce available pairs of input data [32]. Therefore, to enrich our dataset, we adopt an approach similar to recent works (e.g., [36, 99, 20]) on extracting provenance graphs from cyber threat intelligence (CTI) reports such as vulnerability databases [10]. Similar to such solutions, we leverage a combination of rule-based and machine learning techniques (e.g., Part-of-Speech Tagging [26]) to extract different components of provenance graphs (e.g., affected systems, attackers' activities, and the impact of attacks), which allows us to generate a large number of provenance graphs paired with their CTI reports to train our translation model. To this end, we processed 60,000 CVE entries. Inspired by existing solutions (e.g., [99]), to decrease the verbosity of CVE entries and facilitate extracting provenance information, we apply a summarization technique[1] to the entries, and subsequently, extract provenance metadata. Finally, we clean the dataset my removing the entries from which the attackers' activities and impact cannot be extracted, and we obtain six datasets with the total number of 40,151 entries as shown in Table 19. We randomly select 80%, 10% and 10% of entries in each dataset for training, testing and validation, respectively.

Table 19: Statistics of our datasets prepared with CVE entries.

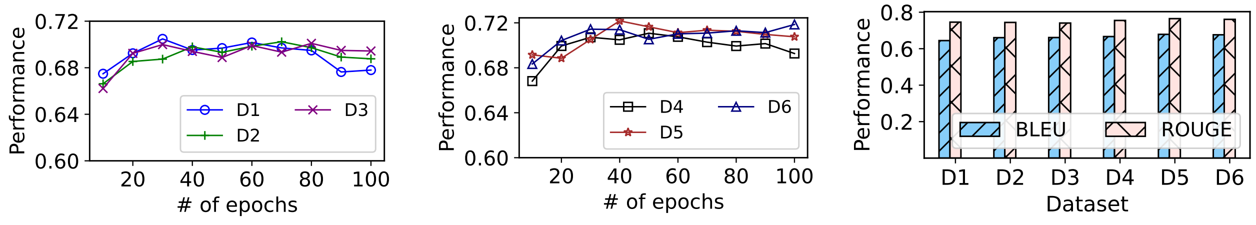
	D_1	D_2	D_3	D_4	D_5	D_6
Total before cleaning	30000	36000	42000	48000	54000	60000
Total after cleaning	20626	25188	28575	32333	36283	40151
Training	16600	20271	22997	26022	29201	32314
Validation	2060	2514	2852	3227	3621	4007
Testing	1966	2403	2726	3084	3461	3830

Number of epochs. We showcase the high quality of our generated reports by first identifying the number of epochs that yields the highest performance (average BLEU and ROUGE scores) for each dataset. Fig. 54(a) shows that VinciDecoder achieves higher performance with smaller datasets after a fewer number of epochs (e.g., 30 epochs for D1). This can be explained by the possibility that training on smaller datasets for a larger number of epochs would cause overfitting [73], which decreases the performance. On the other hand, the performance related to our larger datasets (D4, D5 and D6 in Fig. 54(b)) remains high for a larger number of epochs. For instance, we maximise the performance by training on our largest dataset (D6) for 100 epochs.

Number of training samples. We measure the performance of VinciDecoder trained with different datasets for the number of epochs that achieved the highest performance in Fig. 54(a) and 54(b) (e.g., 30 and 100 epochs for D1 and D6, respectively). Fig. 54(c) shows that both the BLEU and ROUGE scores remain almost similar and above 0.68 and 0.74, respectively, for all datasets. This

[1]`https://pypi.org/project/nlpaug/`

shows that despite the complex content and various writing styles that are natural to CVE reports, VinciDecoder performs well in generating such reports[1].



(a) Performance vs. # of epochs (smaller datasets). (b) Performance vs. # of epochs (larger datasets). (c) Performance vs. size of datasets.

Figure 54: Evaluation with CVE-based provenance graphs.

5.4.3 User-based Study

To evaluate the quality and usefulness of our generated reports, we conduct a user study[2] based on standard practices [14], where participants have to evaluate the factual correctness and fluency of the reports generated by VinciDecoder. Our participants include eight cybersecurity researchers working in a major telecommunication organization and five graduate researchers working in cybersecurity labs of our university. Table 20 shows the percentage of participants in each group, their reported level of expertise, and the average score for all statements.

Table 20: Average quantified agreement levels for each group (scores will be explained later). PG means provenance analysis. (A), (L), and (N) signs represent advanced, little and no knowledge, respectively, as reported by the participants.

	Industry					**Academia**
Background (Cloud-PG)	A-A	A-L	A-N	L-L	L-N	A-L
Participants (%)	15	23	8	8	8	38
Scores (out of 5)	3.83	4.28	3.83	3.83	3	4.13

[1]Note that while both sets of our experiments in Section 5.4.1 and 5.4.2 show high quality reports, directly comparing their results is not meaningful as their reports are of incomparable lengths (e.g., cloud management-level provenance graph-based reports are typically longer which has a negative effect on the performance).

[2]This study has been identified as quality assurance by Research Ethics/Office of Research of our university, which means it requires no ethics approval.

At the beginning of the study, we show an attack scenario (our motivating example in Section 5.1) to the participants. Next, we provide the participants with the provenance graph, the report generated by VinciDecoder, and the manually written report. Our study asks participants to evaluate their investigation with and without VinciDecoder, and accordingly express their level of agreement with the provided statements (shown in Table 21) by choosing one of the following options: *Strongly agree*, *Agree*, *Neutral*, *Disagree* and *Strongly disagree*. We then quantify the results by assigning an integer between one and five to each option, where five means *Strongly agree* and one means *Strongly disagree*.

Table 21: Survey statements and scores. The agreement level of participants are converted to scores between one and five (score five represents *Strongly agree*).

	Statement	**Score**
S1	Understanding attack steps using the generated text is easier than using the path.	4.3
S2	The generated text is consistent with the explained attack scenario.	3.92
S3	The generated text is consistent with the path regarding the relationships of operations.	4.31
S4	The generated text captures all the information of the suspicious path.	4
S5	The generated text is sufficiently fluent compared with the manually written report.	3.46
S6	The generated text is consistent with the manually written report regarding attack steps.	3.92

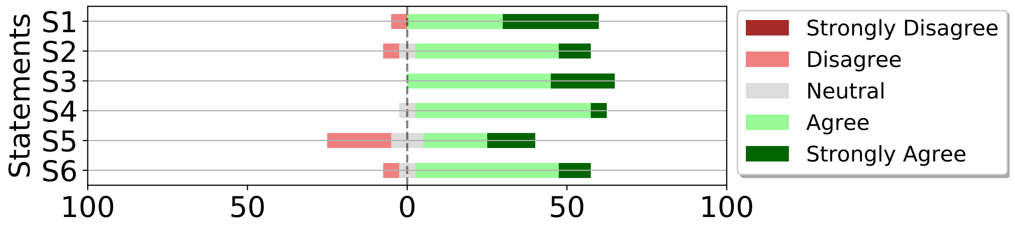


Figure 55: Participants' agreement with statements in Table 21.

Fig. 55 shows the distribution of participants' agreement with each statement. For most participants, understanding the attack steps is much easier using our generated report than the provenance graph (S1). According to most participants, our generated report contains no information contradicting the described attack scenario and the provenance graph (S2 and S3). Additionally, the results (S4) affirm that all the information captured by the provenance graph is reflected in our generated report. Most users find the generated report almost as fluent as the manually created one (S5), while the slightly lower fluency is expected for automatically generated reports [54]. Finally, the attack steps described by the generated report is consistent with the report created by the human analyst (S6). We show the average quantified scores in Table 20. VinciDecoder achieves scores above three among all groups despite their low level (little or no) of expertise, which confirm the benefits of VinciDecoder to users in investigating incidents.

5.5 Discussion

In this section, we discuss future directions and limitations of VinciDecoder.

Application to other models. Our approach is generic enough to support various provenance models (e.g., [49, 72] and [117] for the OS and Internet of things environments, respectively) after converting paths into primitive sentences capturing both nodes and edges as words in our intermediary language. Likewise, an interesting future direction is to apply VinciDecoder to other graphical security models such as attack graphs [20] paired with their corresponding textual interpretation.

Coverage. In this work, we leverage NMT for generating forensic reports from long suspicious paths, as it is known to perform well in translating long sentences [105]. In our future work, we will further investigate the possibility of applying other translation techniques [58] that may increase the performance of VinciDecoder. Finally, our goal is to assist analysts, instead of replacing them, by allowing them to focus on more important but light-weight tasks, e.g., validating the report to ensure its legal value.

5.6 Related Work

Provenance-based security solutions have been extensively explored [49, 92, 91, 117, 106]. King et al. [49] propose data provenance to investigate security incidents in operating systems.

ProvDetector[118] is a provenance solution to detect anomalous programs using embedded sentences representing paths. SteinerLog [19] detects attack campaigns across multiple hosts using alert correlation. Some of the recent solutions (e.g., [109, 72, 125]) focus on increasing the interpretability of provenance graphs. ATLAS [13] adopts sequence learning to model the signature of attacks. In [71, 36, 99], the authors enable threat hunting using the graph capturing the attackers' behavior. Such a graph can be either manually [71] or automatically [36, 99] constructed based on the attackers' behavior that is described by open-source cyber threat intelligence (CTI) reports. There exist efforts adapting provenance analysis to domains other than operating systems such as the Internet of Things (IoT) (e.g., [117]) and SDN environments (e.g., [115, 112]). Wu et al. [121] propose an approach explaining the absence of events. The authors in [59] and [15] propose a provenance-based investigation and access control scheme for clouds, respectively. The authors in [81], propose a solution to enhance the access control mechanism in OpenStack. Chen et. al [25] propose CLARION to capture precise provenance graphs across namespaces of different containers. Unlike our work, none of those solutions generates a human-readable description of the provenance graph, and our approach can be applied to most of those solutions to automatically translate their results into natural language reports.

Several solutions [52, 95, 56] have been proposed to generate human-readable descriptions based on non-linguistic information. The authors in [95] propose a solution to generate textual summaries about basketball games based on tables of information using NMT. [56] is a neural text generation solution to generate the first sentence of a Wikipedia entry based on a provided *infobox*. Finally, [52] proposes a solution that generates abstracts for scientific papers (with the BLEU score of around 0.14) based on paired titles and knowledge graphs (with 4.43 edges, on average). None of those solutions are designed for generating forensic reports based on typically larger and more complex provenance graphs that are natural to the security context or cloud scale. ProvTalk [109] proposes a rule-based approach for generating textual summaries of provenance graphs, which is generalized and complemented with a learning-based approach in VinciDecoder.

5.7 Conclusion

In this chapter, we presented VinciDecoder, the first solution for automatically translating provenance analysis results into human-readable forensic reports using both rule-based and learning-based techniques. To this end, we first explored the characteristics of the provenance graph to represent it in an intermediary language, which can then be translated into a natural language. We showed the feasibility of our approach by implementing VinciDecoder based on an OpenStack cloud, and demonstrated the high quality of generated reports for real-world incident scenarios using both numerical (up to 0.56 and 0.68 BLEU scores for cloud management-level and CVE-based provenance graphs, respectively) and user-based evaluations. As future work, we will integrate VinciDecoder with other (e.g., OS-level) provenance analysis tools. We will also explore other translation techniques and hyperparameters (i.e., the size of embedding vectors and batch size), which may further improve the effectiveness of our approach.

Chapter 6

Other Contributions

Security verification ensures the compliance of cloud environments with specified sets of security properties. There exist solutions that hold the execution of management operations to first verify the compliance of clouds updated by those operations [21]. To reduce the additional delay caused by those verification tasks, proactive solutions (e.g., [66]) initiate an early compliance verification upon predicting suspicious management operations based on the most recent intercepted operation and its frequently subsequent operations. Such an early initiation means that the verification task can be completed by the time the suspicious operation is intercepted, and thus the verification decision (e.g., deny/allow) can be immediately enforced. To this end, my early research in the domain of cloud security [107, 67, 69, 68] contributed to building and presenting such an interception and decision enforcement mechanism, log processing engine, and learning frequent sequences of operations as we detail below.

Runtime API call interception and verification decision enforcement. We first built a middleware for intercepting management API calls and runtime policy enforcement. Additionally, this middleware addresses the key challenge of event type identification by enriching logs with the extracted details of management API calls that are not captured by standard cloud log entries. To build this work, we first studied the design of OpenStack services and chains of Web Server Gateway Interfaces (WSGI) [11], which are basically chains of functions called by each management

API call to conduct a certain task (e.g., authorization) on the received API call and pass it to the next function on the chain, and finally, to the endpoint service. Next, we designed our middleware as an additional WSGI and inserted it into the filter chain corresponding to each managerial service. Additionally, we integrated this work with an existing proactive security verification approach [66] to enable efficient policy compliance verification, and demonstrated its effectiveness in preventing exploits of vulnerabilities through a use case scenario.

Log processor and learning frequent sequences of events. We also built a processor engine for OpenStack cloud logs. To this end, we studied the structure of logs generated by networking and computing services of two different versions of OpenStack that were deployed in our testbed cloud and a real cloud environment hosted by a telecommunication vendor. Specifically, we investigated patterns of different fields in log entries, and identified which component each field represents (e.g., affected resources, cloud tenants requesting the logged operation, type of operations, etc.), and accordingly designed rules for processing log entries and extracting those fields using Logstash [6]. To identify the operations that usually precedes the suspicious ones, we learned frequent sequence of operations by adopting SPMF [34] and applying various pattern mining techniques such as PrefixSpan [93] and MaxSP [35] on our temporally ordered events. Additionally, to enable applying Bayesian Network for learning such preceding operations, we designed a customised algorithm for building the structure of the network. Finally, we implemented a graphical user interface, which can be adopted by the cloud admins to easily monitor the verified security properties and the events that are blocked/warned/allowed based the verification results.

Chapter 7

Conclusion

The increasing attention drawn to cyber attacks in virtualized environments coupled with their high dynamicity and operational complexity highlights the importance of security mechanisms that can be practically applied to such environments. In this thesis, we presented effective and practical provenance-based solutions to facilitate forensic analysis and threat prevention mechanisms in virtualized environments. To this end, we first lift the provenance analysis to the cloud management-level (as opposed to e.g., system call-level) which facilitates the forensic analysis and threat prevention in clouds by tracing configuration changes at a higher abstraction level. We also provided efficient mechanisms to further facilitate the investigation by pruning the provenance graphs and prevent security incidents. Second, we built a system for multi-level provenance analysis in NFV environments. Our system enables tracing back incidents to their root cause operations at different levels of the NFV stack. We also improve the interpretability of multi-level provenance graphs by leveraging the inherent cross-level dependencies in NFV environments. Third, we built a framework to automate the creation of forensic reports by translating the result of provenance analysis into forensic reports.

As future work, we plan to integrate our management-level provenance solutions (i.e., ProvTalk and DominoBlocker) with system-level provenance works to provide mutual benefits (e.g., our solutions can guide system-level solutions to focus on specific resources, while the latter can provide detailed analysis on those resources). Moreover, we envision that the system calls captured within virtual resources (e.g., using SPADE system [38]) can be integrated with the provenance graph generated by our solutions in a manner consistent with previous works (e.g., [74]). Furthermore, we will integrate VinciDecoder with existing approaches for identifying suspicious paths and subgraphs of provenance graphs (e.g., [118]). We will also investigate the effect of using different learning and embedding techniques on the reduction power of ProvTalk, and leverage different translation mechanisms, which may further improve the quality of reports generated by VinciDecoder.

Bibliography

[1] Cisco AVOS, Accessed August 30, 2022. https://github.com/CiscoSystems/avos.

[2] Vitrage RCA service, Accessed August 30, 2022. https://governance.openstack.org/tc/reference/projects/vitrage.html.

[3] Tacker-OpenStack API, Accessed August 30, 2022. https://docs.openstack.org/api-ref/.

[4] OpenStack Congress, Accessed August 30, 2022. https://wiki.openstack.org/wiki/Congress.

[5] Adding malformed security group rule, Accessed February 14, 2022. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-9735.

[6] Logstash, Accessed August 31, 2022. https://www.elastic.co/logstash/.

[7] CVE-2014-0056. Accessed July 28, 2022, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0056/.

[8] CVE-2016-7498. Accessed July 28, 2022, https://nvd.nist.gov/vuln/detail/CVE-2016-7498.

[9] CVE-2020-17376. Accessed July 28, 2022, https://bugs.launchpad.net/nova/+bug/1890501.

[10] CVE details. Accessed June 14, 2022, https://www.cvedetails.com/vulnerability-list/.

[11] Openstack wsgi. Accessed August 30, 2022, Middleware Architecture. https://docs.openstack.org/keystonemiddleware/latest/middlewarearchitecture.

[12] OSSA-2015-018. Accessed August 30, 2022, https://security.openstack.org/ossa/OSSA-2015-018.

[13] Abdulellah Alsaheel, Yuhong Nan, Shiqing Ma, Le Yu, Gregory Walkup, Z Berkay Celik, Xiangyu Zhang, and Dongyan Xu. ATLAS: A Sequence-based Learning Approach for Attack Investigation. In *USENIX Security*, pages 3005–3022, 2021.

[14] Ahlem Assila, Houcine Ezzedine, et al. Standardized usability questionnaires: Features and quality focus. *Electronic Journal of Computer Science and Information Technology: eJCIST*, 2016.

[15] Adam Bates, Benjamin Mood, Masoud Valafar, and Kevin R. B. Butler. Towards Secure Provenance-based Access Control in Cloud Environments. In *CODASPY*, pages 277–284, 2013.

[16] Adam Bates, Dave Jing Tian, Kevin RB Butler, and Thomas Moyer. Trustworthy Whole-System Provenance for the Linux Kernel. In *USENIX Security*, pages 319–334, 2015.

[17] Erick Bauman, Gbadebo Ayoade, and Zhiqiang Lin. A Survey on Hypervisor-based Monitoring: Approaches, Applications, and Evolutions. *ACM Computing Surveys (CSUR)*, 48(1):10, 2015.

[18] Khalid Belhajjame, Reza B'Far, James Cheney, Sam Coppens, Stephen Cresswell, Yolanda Gil, Paul Groth, Graham Klyne, Timothy Lebo, Jim McCusker, et al. PROV-DM: The PROV

Data Model. Technical Report REC-prov-dm-20130430, W3C, 2013. W3C Recommendation. Accessed August 30, 2022. `https://www.w3.org/TR/prov-dm/`.

[19] Bibek Bhattarai and Howie Huang. SteinerLog: Prize Collecting the Audit Logs for Threat Hunting on Enterprise Network. In *ASIA CCS*, pages 97–108, 2022.

[20] Hodaya Binyamini, Ron Bitton, Masaki Inokuchi, Tomohiko Yagyu, Yuval Elovici, and Asaf Shabtai. A Framework for Modeling Cyber Attack Techniques from Security Vulnerability Descriptions. In *KDD*, page 2574–2583, 2021.

[21] Sören Bleikertz, Carsten Vogel, Thomas Groß, and Sebastian Mödersheim. Proactive Security Analysis of Changes in Virtualized Infrastructures. In *ACSAC*, pages 51–60. ACM, 2015.

[22] Ang Chen, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. One primitive to diagnose them all: Architectural support for internet diagnostics. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 374–388, 2017.

[23] Ang Chen, Yang Wu, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. The good, the bad, and the differences: Better network diagnostics with differential provenance. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 115–128, 2016.

[24] Stanley F Chen and Joshua Goodman. An empirical study of smoothing techniques for language modeling. *Computer Speech & Language*, 13(4):359–394, 1999.

[25] Xutong Chen, Hassaan Irshad, Yan Chen, Ashish Gehani, and Vinod Yegneswaran. CLARION: Sound and Clear Provenance Tracking for Microservice Deployments. In *USENIX Security*, pages 3989–4006, 2021.

[26] Alebachew Chiche and Betselot Yitagesu. Part of speech tagging: a systematic review of deep learning and machine learning approaches. *Journal of Big Data*, 9(1):1–25, 2022.

[27] Kyunghyun Cho, Bart van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the Properties of Neural Machine Translation: Encoder–Decoder Approaches. In *SSST*, pages 103–111. ACL, 2014.

[28] Cytoscape. Cytoscape: Open Source Platform for Complex Networks. Accessed August 30, 2022. `https://cytoscape.org/`.

[29] CVE Details. OpenStack Vulnerabilities. Accessed August 30, 2022, `https://www.cvedetails.com/vulnerability-list/vendor_id-11727/Openstack.html`.

[30] Abhishek Dwaraki, Shachi Kumary, and Tilman Wolf. Automated Event Identification from System Logs Using Natural Language Processing. In *2020 International Conference on Computing, Networking and Communications (ICNC))*. IEEE, 2020.

[31] European Telecommunications Standard Institute. Network Functions Virtualisation (NFV); Architectural Framework. Technical Report ETSI GS NFV 002, V1.2.1, 2014.

[32] Marzieh Fadaee, Arianna Bisazza, and Christof Monz. Data Augmentation for Low-Resource Neural Machine Translation. In *ACL*, pages 567–573, 2017.

[33] Matthias Flittner, Johannes M Scheuermann, and Robert Bauer. ChainGuard: Controller-Independent Verification of Service Function Chaining in Cloud Computing. In *NFV-SDN*, pages 1–7. IEEE, 2017.

[34] Philippe Fournier-Viger. SPMF, Accessed August 30, 2022. `http://www.philippe-fournier-viger.com/spmf/index.php`.

[35] Philippe Fournier-Viger, Cheng-Wei Wu, and Vincent S Tseng. Mining maximal sequential patterns without candidate maintenance. In *International Conference on Advanced Data Mining and Applications*, pages 169–180. Springer, 2013.

[36] Peng Gao, Fei Shao, Xiaoyuan Liu, Xusheng Xiao, Zheng Qin, Fengyuan Xu, Prateek Mittal, Sanjeev R Kulkarni, and Dawn Song. Enabling Efficient Cyber Threat Hunting with Cyber Threat Intelligence. In *ICDE*, pages 193–204. IEEE, 2021.

[37] Albert Gatt and Ehud Reiter. SimpleNLG: A Realisation Engine for Practical Applications. In *ENLG*, pages 90–93, 2009.

[38] Ashish Gehani and Dawood Tariq. Spade: Support for provenance auditing in distributed environments. In *Middleware*, 2012.

[39] Xueyuan Han, Thomas Pasquier, Adam Bates, James Mickens, and Margo Seltzer. Unicorn: Runtime Provenance-based Detector for Advanced Persistent Threats. In *NDSS*, 2020.

[40] Xueyuan Han, Thomas Pasquier, Tanvi Ranjan, Mark Goldstein, and Margo Seltzer. FRAPpuccino: Fault-Detection Through Runtime Analysis of Provenance. In *HotCloud*, 2017.

[41] Ragib Hasan, Radu Sion, and Marianne Winslett. The Case of the Fake Picasso: Preventing History Forgery with Secure Provenance. In *FAST*, pages 1–14, 2009.

[42] Wajih Ul Hassan, Lemay Aguse, Nuraini Aguse, Adam Bates, and Thomas Moyer. Towards Scalable Cluster Auditing Through Grammatical Inference over Provenance Graphs. In *NDSS*, 2018.

[43] Wajih Ul Hassan, Shengjian Guo, Ding Li, Zhengzhang Chen, Kangkook Jee, Zhichun Li, and Adam Bates. NoDoze: Combatting Threat Alert Fatigue with Automated Provenance Triage. In *NDSS*, 2019.

[44] Wajih Ul Hassan, Mohammad A Noureddine, Pubali Datta, and Adam Bates. OmegaLog: High-Fidelity Attack Investigation via Transparent Multi-layer Log Analysis. In *NDSS*, 2020.

[45] Red Hat. CVE-2020-12689: Keystone Credential Modification, 2020. Accessed August 30, 2022. `https://access.redhat.com/security/cve/cve-2020-12689/`.

[46] Di He, Hanqing Lu, Yingce Xia, Tao Qin, Liwei Wang, and Tie-Yan Liu. Decoding with Value Networks for Neural Machine Translation. *Advances in Neural Information Processing Systems*, 30, 2017.

[47] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural computation*, 9(8):1735–1780, 1997.

[48] Chris Johnson, Lee Badger, David Waltermire, Julie Snyder, Clem Skorupka, et al. Guide to cyber threat information sharing. *NIST special publication*, 800(150), 2016.

[49] Samuel T. King and Peter M. Chen. Backtracking Intrusions. In *SOSP*, pages 223–236, 2003.

[50] Guillaume Klein, Yoon Kim, Yuntian Deng, Jean Senellart, and Alexander Rush. OpenNMT: Open-Source Toolkit for Neural Machine Translation. In *Proceedings of ACL, System Demonstrations*, pages 67–72. ACL, 2017.

[51] Oldřich Kodym, Lukáš Kubáč, and Libor Kavka. Risks associated with logistics 4.0 and their minimization using blockchain. *Open Engineering*, 10(1):74–85, 2020.

[52] Rik Koncel-Kedziorski, Dhanush Bekal, Yi Luan, Mirella Lapata, and Hannaneh Hajishirzi. Text Generation from Knowledge Graphs with Graph Transformers. In *NAACL*, 2019.

[53] Dan Kushnir and Maayan Goldstein. Causality Inference for Failures in NFV. In *INFOCOM WKSHPS*, pages 929–934. IEEE, 2016.

[54] Samuel Läubli, Rico Sennrich, and Martin Volk. Has Machine Translation Achieved Human Parity? A Case for Document-level Evaluation. In *EMNLP*, pages 4791–4796. ACL, 2018.

[55] Alon Lavie. Evaluating the Output of Machine Translation Systems. *AMTA Tutorial*, 86, 2010.

[56] Rémi Lebret, David Grangier, and Michael Auli. Neural Text Generation from Structured Data with Application to the Biography Domain. In *EMNLP*, pages 1203–1213. ACL, 2016.

[57] Min Li, Wanyu Zang, Kun Bai, Meng Yu, and Peng Liu. MyCloud: Supporting User-Configured Privacy Protection in Cloud Computing. In *ACSAC*, pages 59–68. ACM, 2013.

[58] Adam Lopez. Statistical Machine Translation. *ACM Computing Surveys (CSUR)*, 40(3):1–49, 2008.

[59] Rongxing Lu, Xiaodong Lin, Xiaohui Liang, and Xuemin (Sherman) Shen. Secure Provenance: The Essential of Bread and Butter of Data Forensics in Cloud Computing. In *ASIA CCS*, pages 282–292, 2010.

[60] Yang Luo, Wu Luo, Tian Puyang, Qingni Shen, Anbang Ruan, and Zhonghai Wu. OpenStack Security Modules: A Least-invasive Access Control Framework for the Cloud. In *IEEE CLOUD*, pages 51–58, 2016.

[61] Alexandra L'Heureux, Katarina Grolinger, Hany F. Elyamany, and Miriam A. M. Capretz. Machine learning with big data: Challenges and approaches. *IEEE Access*, 5:7776–7797, 2017.

[62] Shiqing Ma, Juan Zhai, Fei Wang, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. MPI: Multiple Perspective Attack Investigation with Semantic Aware Execution Partitioning. In *USENIX Security*, pages 1111–1128, 2017.

[63] Taous Madi, Yosr Jarraya, Amir Alimohammadifar, Suryadipta Majumdar, Yushun Wang, Makan Pourzandi, Lingyu Wang, and Mourad Debbabi. ISOTOP: Auditing Virtual Networks Isolation Across Cloud Layers in OpenStack. *ACM Transactions on Privacy and Security (TOPS)*, 22(1):1, 2018.

[64] Taous Madi, Mengyuan Zhang, Yosr Jarraya, Amir Alimohammadifar, Makan Pourzandi, Lingyu Wang, and Mourad Debbabi. QuantiC: Distance Metrics for Evaluating Multi-Tenancy Threats in Public Cloud. In *CloudCom*, pages 163–170. IEEE, 2018.

[65] Suryadipta Majumdar, Gagandeep Singh Chawla, Amir Alimohammadifar, Taous Madi, Yosr Jarraya, Makan Pourzandi, Lingyu Wang, and Mourad Debbabi. Prosas: Proactive security auditing system for clouds. *IEEE Transactions on Dependable and Secure Computing*, 2021.

[66] Suryadipta Majumdar, Yosr Jarraya, Momen Oqaily, Amir Alimohammadifar, Makan Pourzandi, Lingyu Wang, and Mourad Debbabi. LeaPS: Learning-based Proactive Security Auditing for Clouds. In *ESORICS*, pages 265–285. Springer, 2017.

[67] Suryadipta Majumdar, Azadeh Tabiban, Yosr Jarraya, Momen Oqaily, Amir Alimohammadifar, Makan Pourzandi, Lingyu Wang, and Mourad Debbabi. Learning Probabilistic Dependencies among Events for Proactive Security Auditing in Clouds. *Journal of Computer Security*, 27(2):165–202, 2019.

[68] Suryadipta Majumdar, Azadeh Tabiban, Meisam Mohammady, Alaa Oqaily, Yosr Jarraya, Makan Pourzandi, Lingyu Wang, and Mourad Debbabi. Multi-level proactive security auditing for clouds. In *DSC*, pages 1–8. IEEE, 2019.

[69] Suryadipta Majumdar, Azadeh Tabiban, Meisam Mohammady, Alaa Oqaily, Yosr Jarraya, Makan Pourzandi, Lingyu Wang, and Mourad Debbabi. Proactivizer: Transforming existing verification tools into efficient solutions for runtime security enforcement. In *European Symposium on Research in Computer Security*, pages 239–262. Springer, 2019.

[70] Hui Miao and Amol Deshpande. Understanding Data Science Lifecycle Provenance via Graph Segmentation and Summarization. In *ICDE*, pages 1710–1713. IEEE, 2019.

[71] Sadegh M Milajerdi, Birhanu Eshete, Rigel Gjomemo, and VN Venkatakrishnan. Poirot: Aligning Attack Behavior with Kernel Audit Records for Cyber Threat Hunting. In *CCS*, pages 1795–1812, 2019.

[72] Sadegh Momeni Milajerdi, Rigel Gjomemo, Birhanu Eshete, R. Sekar, and V. N. Venkatakrishnan. HOLMES: Real-Time APT Detection through Correlation of Suspicious Information Flows. In *IEEE S&P*, pages 1137–1152, 2019.

[73] Tom M Mitchell. *Machine Learning*. McGraw-hill New York, 1997.

[74] Kiran-Kumar Muniswamy-Reddy, Uri Braun, David A. Holland, Peter Macko, Diana L. MacLean, Daniel W. Margo, Margo I. Seltzer, and Robin Smogor. Layering in Provenance Systems. In *USENIX ATC*, 2009.

[75] Kiran-Kumar Muniswamy-Reddy, David A Holland, Uri Braun, and Margo I Seltzer. Provenance-aware Storage Systems. In *USENIX ATC*, pages 43–56, 2006.

[76] Andrés F Murillo, Sandra Julieta Rueda, Laura Victoria Morales, and Álvaro A Cárdenas. SDN and NFV Security: Challenges for Integrated Solutions. In *Guide to Security in SDN and NFV*, pages 75–101. Springer, 2017.

[77] Annamalai Narayanan, Mahinthan Chandramohan, Rajasekar Venkatesan, Lihui Chen, Yang Liu, and Shantanu Jaiswal. graph2vec: Learning Distributed Representations of Graphs. *CoRR*, abs/1707.05005, 2017.

[78] Neo4j. Cypher Query Language. Accessed August 30, 2022. `https://neo4j.com/developer/cypher-query-language/`.

[79] Neo4j. Export to json. Accessed August 30, 2022, `https://neo4j.com/labs/apoc/4.1/export/json/`.

[80] Neo4j. Neo4j Graph Platform. Accessed August 30, 2022. `https://neo4j.com/`.

[81] Dang Nguyen, Jaehong Park, and Ravi Sandhu. Adopting Provenance-based Access Control in OpenStack Cloud IaaS. In *NSS*, pages 15–27. Springer, 2014.

[82] Openstack. NFV API reference. Accessed August 30, 2022. https://developer.openstack.org/api-ref/nfv-orchestration/v1/.

[83] OpenStack. Open Source Cloud Computing Infrastructure. Accessed August 30, 2022. https://www.openstack.org/.

[84] OpenStack. OpenStack Foundation Report, Accelerating NFV Delivery with OpenStack. Technical report. Accessed August 30, 2022. https://object-storage-ca-ymq-1.vexxhost.net/swift/v1/6e4619c416ff4bd19e1c087f27a43eea/www-assets-prod/marketing/OpenStack-NFV-A4.pdf.

[85] OpenStack. OSSA-2014-008: Routers can be cross plugged by other tenants. Accessed August 30, 2022, https://security.openstack.org/ossa/OSSA-2014-008.

[86] OpenStack. OSSA-2015-018: Neutron Firewall Rules Bypass Through Port Update. Accessed August 30, 2022. https://security.openstack.org/ossa/OSSA-2015-018.html.

[87] OpenStack. Service Function Chaining Extension for OpenStack. Accessed August 30, 2022. https://docs.openstack.org/networking-sfc/latest/.

[88] OpenStack. Tacker Documentation. Accessed August 30, 2022. https://docs.openstack.org/tacker/.

[89] OpenStack. Kubernetes as VIM in Tacker, 2021. Accessed August 30, 2022. https://specs.openstack.org/openstack/tacker-specs/specs/queens/Kubernetes-as-VIM.html.

[90] Opentsack. API Reference. Accessed August 30, 2022. https://developer.openstack.org/api-ref/.

[91] Thomas Pasquier, Xueyuan Han, Mark Goldstein, Thomas Moyer, David Eyers, Margo Seltzer, and Jean Bacon. Practical Whole-System Provenance Capture. In *SoCC*, pages 405–418. ACM, 2017.

[92] Thomas Pasquier, Xueyuan Han, Thomas Moyer, Adam Bates, Olivier Hermant, David Eyers, Jean Bacon, and Margo Seltzer. Runtime Analysis of Whole-System Provenance. In *CCS*, pages 1601–1616. ACM, 2018.

[93] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, Jianyong Wang, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Mei-Chun Hsu. Mining sequential patterns by pattern-growth: The prefixspan approach. *IEEE Transactions on knowledge and data engineering*, 16(11):1424–1440, 2004.

[94] Devin J. Pohly, Stephen E. McLaughlin, Patrick D. McDaniel, and Kevin R. B. Butler. Hi-Fi: Collecting High-Fidelity Whole-System Provenance. In *ACSAC*, pages 259–268, 2012.

[95] Ratish Puduppully, Li Dong, and Mirella Lapata. Data-to-text generation with content selection and planning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, pages 6908–6915, 2019.

[96] François Reynaud, François-Xavier Aguessy, Olivier Bettan, Mathieu Bouet, and Vania Conan. Attacks against Network Functions Virtualization and Software-Defined Networking: State-of-the-art. In *NetSoft*, pages 471–476. IEEE, 2016.

[97] Areeg Samir and Claus Pahl. 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.

[98] Manuel A Borroto Santana, Francesco Ricca, and Bernardo Cuteri. Reducing the Impact of out of Vocabulary Words in the Translation of Natural Language Questions into SPARQL Queries. *arXiv preprint arXiv:2111.03000*, 2021.

[99] Kiavash Satvat, Rigel Gjomemo, and VN Venkatakrishnan. EXTRACTOR: Extracting Attack Behavior from Threat Reports. In *EuroS&P*, pages 598–615. IEEE, 2021.

[100] Carla Sauvanaud, Kahina Lazri, Mohamed Kaâniche, and Karama Kanoun. Anomaly Detection and Root Cause Localization in Virtual Network Functions. In *ISSRE*, pages 196–206. IEEE, 2016.

[101] Nabil Schear, Patrick T Cable II, Thomas M Moyer, Bryan Richard, and Robert Rudd. Bootstrapping and Maintaining Trust in the Cloud. In *ACSAC*, pages 65–77. ACM, 2016.

[102] A. W. Services. Mapping AWS Services to the NFV Framework, 2021. Accessed August 30, 2022. `https://aws.amazon.com/cloudformation/`.

[103] Amazon Web Services. Amazon virtual private cloud. Accessed August 30, 2022, `https://aws.amazon.com/vpc`.

[104] Shikhar Sharma, Layla El Asri, Hannes Schulz, and Jeremie Zumer. Relevance of Unsupervised Metrics in Task-Oriented Dialogue for Evaluating Natural Language Generation. *CoRR*, abs/1706.09799, 2017.

[105] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to Sequence Learning with Neural Networks. *Advances in neural information processing systems*, 27, 2014.

[106] Azadeh Tabiban, Yosr Jarraya, Mengyuan Zhang, Makan Pourzandi, Lingyu Wang, and Mourad Debbabi. Catching Falling Dominoes: Cloud Management-Level Provenance Analysis with Application to OpenStack. In *CNS*, pages 1–9. IEEE, 2020.

[107] Azadeh Tabiban, Suryadipta Majumdar, Lingyu Wang, and Mourad Debbabi. PERMON: An Openstack Middleware for Runtime Security Policy Enforcement in Clouds. In *SPC*, pages 1–7. IEEE, 2018.

[108] Azadeh Tabiban, Heyang Zhao, Yosr Jarraya, Makan Pourzandi, and Lingyu Wang. VinciDecoder:Automatically Interpreting Provenance Graphs into Textual Forensic Reports with Application to OpenStack. In *NordSec*, 2022.

[109] Azadeh Tabiban, Heyang Zhao, Yosr Jarraya, Makan Pourzandi, Mengyuan Zhang, and Lingyu Wang. ProvTalk: Towards Interpretable Multi-level Provenance Analysis in Networking Functions Virtualization (NFV). In *NDSS*, 2022.

[110] Sudershan Lakshmanan Thirunavukkarasu, Mengyuan Zhang, Alaa Oqaily, Gagandeep Singh Chawla, Lingyu Wang, Makan Pourzandi, and Mourad Debbabi. Modeling NFV Deployment to Identify the Cross-level Inconsistency Vulnerabilities. In *CloudCom*, pages 167–174. IEEE, 2019.

[111] Brendan Tschaen, Ying Zhang, Theo Benson, Sujata Banerjee, Jeongkeun Lee, and Joon-Myung Kang. SFC-Checker: Checking the Correct Forwarding Behavior of Service Function Chaining. In *NFV-SDN*, pages 134–140. IEEE, 2016.

[112] Benjamin E Ujcich, Samuel Jero, Anne Edmundson, Qi Wang, Richard Skowyra, James Landry, Adam Bates, William H Sanders, Cristina Nita-Rotaru, and Hamed Okhravi. Cross-App Poisoning in Software-Defined Networking. In *CCS*, pages 648–663. ACM, 2018.

[113] Verizon. Verizon Network Infrastructure Planning. Technical report, 2016. Accessed August 30, 2022. `https://m.iotone.com/files/pdf/vendor/Verizon_SDN-NFV_Reference_Architecture.pdf`.

[114] VMware. VMware vSphere, 2020. Accessed August 30, 2022. `https://www.vmware.com/ca/products/vsphere.html`.

[115] Haopei Wang, Guangliang Yang, Phakpoom Chinprutthiwong, Lei Xu, Yangyong Zhang, and Guofei Gu. Towards Fine-grained Network Security Forensics and Diagnosis in the SDN Era. In *CCS*, pages 3–16. ACM, 2018.

[116] Jianyong Wang and Jiawei Han. BIDE: Efficient Mining of Frequent Closed Sequences. In *ICDE*, pages 79–90. IEEE, 2004.

[117] Qi Wang, Wajih Ul Hassan, Adam Bates, and Carl Gunter. Fear and Logging in the Internet of Things. In *NDSS*, 2018.

[118] Qi Wang, Wajih Ul Hassan, Ding Li, Kangkook Jee, Xiao Yu, Kexuan Zou, Junghwan Rhee, Zhengzhang Chen, Wei Cheng, C Gunter, et al. You Are What You Do: Hunting Stealthy Malware via Data Provenance Analysis. In *NDSS*, 2020.

[119] Yushun Wang, Taous Madi, Suryadipta Majumdar, Yosr Jarraya, Amir Alimohammadifar, Makan Pourzandi, Lingyu Wang, and Mourad Debbabi. TenantGuard: Scalable Runtime Verification of Cloud-Wide VM-Level Network Isolation. In *NDSS*, 2017.

[120] WSGI. Python WSGI Middleware. Accessed August 30, 2022, `https://wsgi.readthedocs.io/en/latest/libraries.html`.

[121] Yang Wu, Mingchen Zhao, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. Diagnosing Missing Events in Distributed Systems with Negative Provenance. In *ACM SIGCOMM*, pages 383–394, 2014.

[122] Salifu Yusif and Abdul Hafeez-Baig. A Conceptual Model for Cybersecurity Governance. *Journal of Applied Security Research*, 16(4):490–513, 2021.

[123] Michael Zamot. Accessed August 30, 2022, Where does OpenStack fit in a public cloud world? `https://opensource.com/article/18/3/openstack-public-cloud-world`.

[124] Jun Zeng, Zheng Leong Chua, Yinfang Chen, Kaihang Ji, Zhenkai Liang, and Jian Mao. Watson: Abstracting behaviors from audit logs via aggregation of contextual semantics. In *NDSS*, 2021.

[125] Jun Zeng, Zheng Leong Chua, Yinfang Chen, Kaihang Ji, Zhenkai Liang, and Jian Mao. WATSON: Abstracting Behaviors from Audit Logs via Aggregation of Contextual Semantics. In *NDSS*, 2021.

[126] Xiaoli Zhang, Qi Li, Jianping Wu, and Jiahai Yang. Generic and Agile Service Function Chain Verification on Cloud. In *IWQoS*, pages 1–10. IEEE/ACM, 2017.