INTERPARTY VISIBILITY IN A CLOUD COMPUTING PLATFORM

by

Hyun-wook Baek

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing
The University of Utah
December 2018
The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of Hyun-wook Baek

has been approved by the following supervisory committee members:

   Jacobus Van der Merwe, Chair(s)  25 May 2018
   Date Approved

   Feifei Li, Member  25 May 2018
   Date Approved

   Eric Eide, Member  25 May 2018
   Date Approved

   Ryan Stutsman, Member  25 May 2018
   Date Approved

   Abhinav Srivastava, Member  30 May 2018
   Date Approved

by Ross Whitaker, Chair/Dean of
the Department/College/School of Computing

and by David B. Kieda, Dean of The Graduate School.
ABSTRACT

In recent years, it has become commonplace for organizations to deploy their services in a cloud environment. However, as this new computation ecosystem matured, the unique challenges of it also started to emerge. Due to the inherent multiparty, multilayer environment of public clouds, both cloud providers and cloud tenants have limited visibility into the whole system. This limited visibility complicates problems involving multiple parties in a cloud, such as accounting resource usage, resource demand estimation, and security monitoring for cloud providers as well as troubleshooting, virtual resource performance estimation/optimization, and automating service deployment for cloud tenants. In existing cloud platforms, solutions for these problems often require time-consuming and expensive interaction between different parties. Given that resource optimization and cost saving are becoming top priorities not only for cloud providers but also for tenants, resolving this lack of visibility problem is becoming a critical challenge of cloud computing platforms. However, extending visibility is not a simple task since, for security and privacy, cloud platforms were originally designed to reduce interparty visibility. In addition, for cloud providers, it could be prohibitively expensive to support such extended visibility for every single tenant.

In this dissertation, we study different aspects related to enhancing visibility in a multitenant cloud environment. In the first part of the dissertation, we present a framework to offer cloud tenants better visibility into the cloud infrastructure for better understanding and troubleshooting, where both cloud providers and tenants may save cloud management costs. In the second part, we focus on the cloud provider’s visibility into the tenants’ network traffic, and enhance it by adopting a widely used traffic matrix estimation model for ISP networks and addressing two key challenges to apply the model to datacenter networks: the sparse traffic matrix problem and the interior traffic sink/source problem. In the last part, we study unintended ways to enhance visibility of tenants in the cloud infrastructure. We focus on cloud tenants’ visibility into the states of the virtual firewalls, which are typically supposed to be unknown to cloud tenants, devise a novel method to monitor the states of firewalls, and exploit them as a side channel into the host machine’s infrastructure level activities.
CONTENTS

ABSTRACT ................................................................. iii
LIST OF FIGURES .......................................................... vii
LIST OF TABLES ............................................................... ix
ACKNOWLEDGEMENTS ......................................................... x

CHAPTERS

1. INTRODUCTION ............................................................... 1
   1.1 Dissertation Statement ............................................... 2
   1.2 CloudSight: Transparency as a Cloud Service ...................... 2
   1.3 Polygravity: Multitenant Datacenter Network Tomography .......... 3
   1.4 Exploiting Virtual Firewalls as a Side Channel ..................... 4

2. TENANT-SIDE VISIBILITY INTO THE INFRASTRUCTURE ............... 6
   2.1 Introduction .......................................................... 6
   2.2 Motivation ............................................................ 8
   2.3 CloudSight Architecture ............................................ 10
      2.3.1 Instrumentation and Log-Preprocessing ....................... 11
      2.3.2 CloudSight Resource Graph .................................. 12
      2.3.3 Key Clustering .................................................. 15
   2.4 Implementation .................................................... 18
      2.4.1 Instrumentation ................................................ 18
      2.4.2 Unit Test ....................................................... 18
      2.4.3 Platform-Independent Components .......................... 20
   2.5 Applications ....................................................... 20
      2.5.1 Time-Traveling Cloud Debugger ............................... 20
      2.5.2 Functional Consistency Verifier ............................. 21
      2.5.3 VM Co-residency Check ..................................... 21
      2.5.4 VM Unreachable ............................................... 22
      2.5.5 Firewall Rules Not Enforced ................................ 24
   2.6 Evaluation ......................................................... 25
      2.6.1 Key Clustering Robustness .................................. 25
      2.6.2 Resource Graph Accuracy ................................... 27
      2.6.3 CloudSight Overhead ......................................... 28
      2.6.4 Scalability ...................................................... 28
      2.6.5 Graph Processing Latency .................................. 29
   2.7 Related Work ....................................................... 29
      2.7.1 Tenant Domain Tools ....................................... 29
      2.7.2 Provider Domain Tools ..................................... 30
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.5</td>
<td>Covert Channels</td>
<td>80</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Single-Node Covert Channel</td>
<td>81</td>
</tr>
<tr>
<td>4.5.1.1</td>
<td>Threat Model</td>
<td>81</td>
</tr>
<tr>
<td>4.5.1.2</td>
<td>Covert Channel Mechanism</td>
<td>81</td>
</tr>
<tr>
<td>4.5.1.3</td>
<td>Implementation</td>
<td>82</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Multinode Covert Channel</td>
<td>82</td>
</tr>
<tr>
<td>4.5.2.1</td>
<td>Threat Model</td>
<td>82</td>
</tr>
<tr>
<td>4.5.2.2</td>
<td>Covert Channel Mechanism</td>
<td>83</td>
</tr>
<tr>
<td>4.5.2.3</td>
<td>Implementation</td>
<td>84</td>
</tr>
<tr>
<td>4.6</td>
<td>Evaluation</td>
<td>84</td>
</tr>
<tr>
<td>4.6.1</td>
<td>Accuracy of <em>EpochMonitor</em></td>
<td>84</td>
</tr>
<tr>
<td>4.6.2</td>
<td>Single-Node Covert Channel</td>
<td>84</td>
</tr>
<tr>
<td>4.6.3</td>
<td>Multinode Covert Channel</td>
<td>85</td>
</tr>
<tr>
<td>4.6.4</td>
<td>Side Channel Attack</td>
<td>86</td>
</tr>
<tr>
<td>4.7</td>
<td>Mitigation Techniques</td>
<td>87</td>
</tr>
<tr>
<td>4.7.1</td>
<td>Adjusting the Polling Interval</td>
<td>87</td>
</tr>
<tr>
<td>4.7.2</td>
<td>Request Rate Limiting</td>
<td>88</td>
</tr>
<tr>
<td>4.8</td>
<td>Deployment</td>
<td>88</td>
</tr>
<tr>
<td>4.8.1</td>
<td>Deployment Environment</td>
<td>88</td>
</tr>
<tr>
<td>4.8.1.1</td>
<td>With Helper Nodes</td>
<td>88</td>
</tr>
<tr>
<td>4.8.1.2</td>
<td>Without Helper Nodes</td>
<td>88</td>
</tr>
<tr>
<td>4.8.2</td>
<td>Protocols and Rules</td>
<td>90</td>
</tr>
<tr>
<td>4.8.2.1</td>
<td>ICMP Ping</td>
<td>91</td>
</tr>
<tr>
<td>4.8.2.2</td>
<td>ICMP Boomerang Packets</td>
<td>92</td>
</tr>
<tr>
<td>4.8.2.3</td>
<td>TCP Boomerang Packets</td>
<td>94</td>
</tr>
<tr>
<td>4.8.2.4</td>
<td>UDP Boomerang Packets</td>
<td>95</td>
</tr>
<tr>
<td>4.8.2.5</td>
<td>Other Protocols</td>
<td>95</td>
</tr>
<tr>
<td>4.8.2.6</td>
<td>Ingress Rules or Egress Rules?</td>
<td>96</td>
</tr>
<tr>
<td>4.8.2.7</td>
<td>Probing Deletion of Rules</td>
<td>97</td>
</tr>
<tr>
<td>4.9</td>
<td>Related Work</td>
<td>98</td>
</tr>
<tr>
<td>4.9.1</td>
<td>Information Leakage Channels in Cloud</td>
<td>98</td>
</tr>
<tr>
<td>4.9.2</td>
<td>Shared Services and Distributed Resource Management</td>
<td>98</td>
</tr>
<tr>
<td>4.10</td>
<td>Conclusion and Future Work</td>
<td>99</td>
</tr>
</tbody>
</table>

5. CONCLUSION .......................................................... 100

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Summary of the Dissertation</td>
<td>100</td>
</tr>
<tr>
<td>5.2</td>
<td>Future Research Directions</td>
<td>101</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Cooperative Cloud Troubleshooting</td>
<td>101</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Polygravity for Very Large Scale Datacenter Network</td>
<td>103</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Further Exploitation of Shared Software Resource</td>
<td>103</td>
</tr>
</tbody>
</table>

REFERENCES .......................................................... 105
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>CloudSight architecture.</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>CloudSight operation.</td>
<td>11</td>
</tr>
<tr>
<td>2.3</td>
<td>Log-to-graph conversion</td>
<td>13</td>
</tr>
<tr>
<td>2.4</td>
<td>Capturing relationships between cloud resources.</td>
<td>13</td>
</tr>
<tr>
<td>2.5</td>
<td>Resource graph built by a series of events.</td>
<td>14</td>
</tr>
<tr>
<td>2.6</td>
<td>Synonym clusters to update the schema.</td>
<td>18</td>
</tr>
<tr>
<td>2.7</td>
<td>Security group active logger in OpenStack.</td>
<td>19</td>
</tr>
<tr>
<td>2.8</td>
<td>Inaccurate key clustering scenarios.</td>
<td>26</td>
</tr>
<tr>
<td>2.9</td>
<td>Evaluation results of CloudSight.</td>
<td>27</td>
</tr>
<tr>
<td>3.1</td>
<td>A view of gravity model to an example network.</td>
<td>34</td>
</tr>
<tr>
<td>3.2</td>
<td>Polygravity operation.</td>
<td>37</td>
</tr>
<tr>
<td>3.3</td>
<td>Example Inner Gravity modeling for an interior network device acting as a sink</td>
<td>40</td>
</tr>
<tr>
<td>3.4</td>
<td>An example walk-through of gravity model customization for a cloud network.</td>
<td>41</td>
</tr>
<tr>
<td>3.5</td>
<td>Traffic matrices of tenants</td>
<td>48</td>
</tr>
<tr>
<td>3.6</td>
<td>Cumulative distribution of flows.</td>
<td>49</td>
</tr>
<tr>
<td>3.7</td>
<td>Different levels of access control setups for web service</td>
<td>50</td>
</tr>
<tr>
<td>3.8</td>
<td>Sample traffic matrices in a datacenter with 5 web service tenants and the affinity placement policy</td>
<td>52</td>
</tr>
<tr>
<td>3.9</td>
<td>Sample traffic matrices in a datacenter with 5 map-reduce tenants and the affinity placement policy</td>
<td>53</td>
</tr>
<tr>
<td>3.10</td>
<td>RMSRE of Polygravity with and without IGE and broadcast noise cancellation (N/C).</td>
<td>56</td>
</tr>
<tr>
<td>3.11</td>
<td>The computation time of tomogravity and Polygravity as the target datacenter scales up.</td>
<td>58</td>
</tr>
<tr>
<td>4.1</td>
<td>Resource sharing of two requests</td>
<td>63</td>
</tr>
<tr>
<td>4.2</td>
<td>Timeline of the RPC loop and Epochs</td>
<td>66</td>
</tr>
<tr>
<td>4.3</td>
<td>Various impacts of requests on Epochs</td>
<td>67</td>
</tr>
<tr>
<td>4.4</td>
<td>Average and standard deviation of elapsed time for requests and their combinations</td>
<td>68</td>
</tr>
<tr>
<td>4.5</td>
<td>Execution durations of the RPC loop iterations while add+atch and del+dtch requests are arriving</td>
<td>70</td>
</tr>
<tr>
<td>4.6</td>
<td>Average and standard deviation (over 100 runs) of elapsed time for add and del requests as the number of security group rules is increased.</td>
<td>71</td>
</tr>
</tbody>
</table>
4.7 Average and standard deviation (over 100 runs) of elapsed time for \texttt{atch} and \texttt{dtch} requests as the number of security group rules is increased. ........................................ 71
4.8 Average and standard deviation (over 100 runs) of elapsed time for \texttt{add} request as the number of security group rules used by the long-term impact request increases. . . 72
4.9 Monitoring a virtual firewall. ................................................................. 74
4.10 Timeline of a \texttt{UPDATE+PROBE} step. .................................................. 75
4.11 \textit{n}-Reactive \texttt{UPDATE+PROBE} method ........................................... 77
4.12 Architecture of \textit{EpochMonitor} .......................................................... 79
4.13 Covert channel setups ............................................................................. 81
4.14 Execution durations of the RPC loop while sending \texttt{"hello"} through the \textit{single-node covert channel}. .......................................................... 85
4.15 Execution durations of the RPC loop while sending \texttt{"hello"} through the \textit{multinode covert channel}. .......................................................... 86
4.16 Execution durations of the RPC loop iterations while a VM is monitoring the infrastructure level activities using EpochMonitor ........................................ 87
4.17 Snapshot of example Iptables chains of different OpenStack ..................... 94
LIST OF TABLES

2.1 States of deleted security group rules ................................. 9
2.2 The OpenStack functions hooked by CloudSight. ..................... 19
2.3 Cloud debugger commands. ............................................. 21
3.1 Performance of various algorithms ..................................... 52
3.2 Performance of Polygravity with heterogeneous types of tenants ................................. 54
3.3 The counts of datacenter components as the datacenter scales up ...................... 57
3.4 Measurement error tolerance of Polygravity ............................ 59
4.1 Protocols and rules for probing ......................................... 96
ACKNOWLEDGEMENTS

I would like to thank heavenly Father who has sincerely guided me and trained me through this time. I hope I can serve others with what I have learned, as He taught me through this journey.

Thanks to my beloved wife who walked with me during the journey and always encouraged me to stand up again and again. I also give many thanks to my parents who always trusted me, felt proud of me, and encouraged me through lots of love and prayer. I want to thank my sister and brother-in-law as well as their daughter, who are my fellow workers and offer me an example of faithful life. Special thanks to my former roommate, Changsu Lee, who helped me never forget how to smile in all difficult moments.

I sincerely appreciate the guidance and support of my advisor, Jacobus Van der Merwe. He was one of the greatest teachers in my life and showed an example of a good scholar through his life. I especially thank him for his patience and encouragement, which allowed me to keep moving forward during the most difficult stages of my PhD journey.

In addition, I would like to thank Abhinav Srivastava, Eric Eide, Robert Ricci, Ryan Stutsman, and Feifei Li for feedback on works that are part of this dissertation. It would have been impossible for this dissertation to be completed like this without their advice and the discussions with them. I give special thanks to Eric Eide, who carefully reviewed this manuscript to the end.

I also thank all my colleagues at the Flux Research Group for all their help. Especially, I would like to thank Richard Li and Junguk Cho, who stayed over many nights together and helped me to keep enthusiasm for research. Also, special thanks to David Hancock, who helped me to be a better writer, and Min Du and Anand Tripathi, who gave me valuable advice for applying machine learning techniques.

The work described in this dissertation was supported in part by the National Science Foundation under grant number 1314945.
CHAPTER 1

INTRODUCTION

Due to the inherent multiparty and multilayer environment of public clouds, no single party of a cloud — neither cloud providers nor cloud tenants — has a complete system-wide view. This limited visibility complicates problems involving multiple parties in a cloud, such as accounting resource usage, resource demand estimation, and security monitoring for cloud providers [13, 81]; and troubleshooting, virtual resource performance estimation/optimization, and automating service deployment for cloud tenants [16, 51]. In existing cloud platforms, solutions for these problems often require time-consuming and expensive interaction between different parties. Given that resource optimization and cost saving are becoming top priorities not only for cloud providers but also for tenants [76], resolving the visibility problems is becoming a critical challenge of cloud computing platforms.

A straightforward solution to address these concerns is to extend each party’s visibility into the others by collecting events and metrics of the whole cloud system and delivering them to each party. However, extending visibility is not a simple task since, for security and privacy, cloud platforms were originally designed to reduce interparty visibility — from tenants to provider, from provider to tenants, and intertenant. In addition, for cloud providers, it could be prohibitively expensive to support such extended visibility for every single tenant.

In this dissertation, we study different aspects associated with enhancing visibility in a multitenant cloud environment. In Chapter 2, we first address the visibility problem from the perspective of cloud tenants through a visibility-supporting framework from which both cloud providers and tenants can benefit. In Chapter 3, we address a network-level visibility problem from the cloud provider’s side to allow the cloud provider to obtain visibility of the load imposed by tenants. Finally, in Chapter 4, we revisit the tenant-side visibility problem and explore how unintended visibility into the cloud platform can be obtained and exploited.
1.1 Dissertation Statement

Interparty visibility in a cloud platform can be enhanced either in a symbiotic way, to cooperatively address cloud problems involving multiple parties, or in an uncooperative way, to exploit it.

In what follows, we provide an overview of the works.

1.2 CloudSight: Transparency as a Cloud Service

Since today’s IaaS cloud software systems are under active development, it is difficult to avoid the introduction of bugs. The problem is it is neither the cloud developer nor the cloud provider, but rather the cloud tenant, who most often encounters new bugs. This problem is exacerbated because cloud tenants have almost no means to deal with an infrastructure-level bug. Various cloud monitoring tools are being offered to cloud tenants, but no single tool can clearly indicate whether the cause of a trouble is from the provider’s or the tenant’s side. The fundamental reason for this problem is the way today’s IaaS cloud platforms present an abstract view of the states of resources to tenants. IaaS cloud platforms simplify the work of cloud tenants by providing clean virtual resource abstractions. Specifically, IaaS cloud platforms maintain the states of resources in a central database and forward the information to tenants when tenants query the platform. The complication is that the states recorded in the database are not necessarily identical to the real-world states of resources, either because the actual process of creating/deleting/modifying the resource is not yet complete, or because of a bug or misconfiguration of the system. Regardless, for cloud tenants, there is no way to identify such inconsistency, and this leads to uncertainty with respect to the cause when troubles occur. For a clear answer, tenants must depend on support from the provider, which is costly for both tenant and provider.

Explicitly telling the tenants about inconsistencies can help the tenants help themselves. In other words, if a tenant can distinguish the state of a resource written in the central database from its state directly monitored in the cloud infrastructure and/or its state according to a unit tester, the tenant can diagnose problems of the resource by himself. Under this motivation, we implemented CloudSight, a transparency-as-a-service framework that maintains the history of states of cloud resources from various vantage points. CloudSight dynamically inserts monitoring functions into the target IaaS system to monitor state changes of resources in different vantage points, stitches together state information of the same resource monitored in different places, and maintains the state change history in a graph database. The fact that IaaS software systems are under active development introduces a
significant challenge in the system design of CloudSight: compatibility. Thus, if a tool is made by simply understanding a specific version of the target system and modifying the target system, the amount of time and effort to maintain compatibility with future versions of the IaaS system becomes significant. We combine two key techniques to resolve this challenge. First, CloudSight uses an entity resolution technique named *key clustering* to automate updating domain knowledge of the target system. Second, *dynamic code injection* permits CloudSight to inject monitoring functions into the target system without modifying the target system’s source code. We demonstrated the utility of CloudSight by diagnosing real problems in OpenStack, one of which is a critical security bug first identified by this work.

### 1.3 Polygravity: Multitenant Datacenter Network Tomography

*Network traffic accountability* allows network operators to break down network usage and map it to consumers such as servers, virtual machines, virtual networks, or tenants. Fine-grained network accountability, such as determining the bandwidth of each *point-to-point* flow, is critical to ensure reliable cloud performance and customer satisfaction. For example, when a network hot spot appears, operators wish to determine which consumer is utilizing the network more than it should, or which consumers may suffer performance degradation due to the hot spot. However, network accountability is challenging in multitenant datacenters, as today’s network systems cannot provide line-rate flow measurement for datacenter-scale traffic. An alternative solution to revealing fine-grained flow usage relies on combining the network routing matrix with coarse-grained link-level measurements, e.g., SNMP. Initially developed for ISP networks, *tomogravity* was also applied to datacenters, albeit with less accurate results (e.g., with best-case average relative error around 15%). The main challenge for accurate traffic estimation with tomogravity in a datacenter is that datacenter traffic, naively interpreted, might not exhibit the inherent structure needed for this approach. Specifically, tomogravity was designed with two assumptions, which hold for ISP networks, about the network traffic characteristics: 1) *all nodes proportionally contribute to overall traffic flows*; and 2) *the network does not have internal sinks/sources*. Contrary to the first assumption, recent studies showed that the datacenter traffic matrix is sparse, and all nodes do not contribute to the traffic matrix. In addition, contrary to the second assumption, datacenter networks may contain a variety of internal sinks and sources.

Our key insights to overcome the challenge are as follows. First, datacenter administrators
have access to readily available information about the contributing nodes, e.g., tenant-level virtual
topology configuration or access control setups, which can be used to deal with fact that datacenter
traffic flow is contributed by a limited number of nodes. Second, noise due to internal sinks/sources
can be effectively canceled out by integrating information about their behavior into the tomogravity
model. We devised a new network traffic estimation method named Polygravity, which utilizes such
cloud configuration domain knowledge to enable precise traffic estimation. Polygravity performs
significantly better than previous methods for datacenter network accountability. For tenants with
fine-grained domain knowledge, Polygravity reduces the average relative error of estimating flow
usage to less than 1%. For tenants with coarse-grained domain knowledge, with the assistance of
host-based partial sampling, Polygravity consistently reduces the relative error by \( \frac{1}{3} \) compared to the
relative error of the sampling-only solution.

### 1.4 Exploiting Virtual Firewalls as a Side Channel

Resource sharing is a fundamental part of cloud computing. By multiplexing virtual resources
(virtual machines, virtual networks, virtual firewalls, etc.) across an infrastructure, a cloud provider
maximizes resource utilization of the infrastructure and offers cloud users flexible scaling of virtual
environments with minimal costs. However, shared resources also cause interference among cloud
tenants and can even be exploited as side channels by malicious users to make critical security
breaches. For example, if an attacker’s virtual machine can be successfully placed on a physical
machine hosting targeted virtual machines, the attacker virtual machine can utilize such side channel
mechanisms to detect if it is co-resident with the targets, to degrade performance of the targets
or to break the virtual isolation and steal confidential information from compromised and non-
compromised targets.

The underlying mechanisms exploited by previously studied side-channel attacks were mostly
limited to hypervisor level mechanisms managing a specific set of hardware resources: CPU,
L2/L3 caches, memory, and network devices. However, cloud tenants share resources not only
at the hypervisor level but also at the cloud management level such as processes, threads, kernel
modules, and networks of cloud management systems. Especially in cloud management planes, it is
commonplace for a single software instance to process multiple requests from different tenants (both
at the central cloud controllers and at the distributed cloud management components). For instance,
for two co-residing virtual machines, if each of their users makes a request to connect each virtual
machine to a virtual network, the two requests will go through the same virtual network management software instances, such as a local cloud management service, a local OpenVSwitch service, and a netfilter kernel module. These requests may share some parts of their execution paths. Therefore, the processing times of the two requests may influence each other. Our key insight is that, if a virtual machine can keep track of processing times of its own infrastructure-level requests, it can obtain footprints of co-resident virtual machines’ infrastructure level information — e.g., virtual firewall update time or start/end time of co-resident virtual machines. Since this type of information is not obtainable by previously studied side channels, the potential impact of this new type of side channel can be significant. We devised a new technique to exploit a cloud management-level mechanism as a side channel. We exploited virtual firewalls (so-called “security groups”) as a key medium for the side channel. We demonstrated the utility of the side channel by implementing two different classes of covert channels based on the firewall side channel.
CHAPTER 2

TENANT-SIDE VISIBILITY INTO THE INFRASTRUCTURE

This chapter presents CloudSight, a transparency-as-a-service framework that resolves the visibility problem from the perspective of cloud tenants.¹

2.1 Introduction

Today’s IaaS cloud software systems are under active development. To survive in the cloud marketplace, different software vendors competitively introduce new features. Developing new features over short development cycles for a cloud platform, which is essentially a sophisticated distributed system, has the potential to introduce bugs, which might impact the reliable operation of the system. The problem is, in a cloud environment, it is neither the cloud developer nor the cloud provider, but rather the cloud tenant who most often encounter new bugs.

This problem is exacerbated because cloud tenants have almost no means to deal with an infrastructure level bug. Various cloud monitoring tools are being offered to cloud tenants, but no single tool can clearly indicate whether the cause of a problem is from the provider’s or the tenant’s side. For example, cloud tenants have a difficult time getting answers to questions such as:

• “I have recently rebooted my VM, but cannot access it after a while. Is it because I have mis-configured my VM or because the provider’s server is down?”
• “I have created a firewall, but the forbidden packets are still bypassing it. Is the firewall not actually created? Did I make a mistake? Or is it a system bug?”
• “I created a virtual interface that has silently disappeared. When was it deleted? Did somebody else delete my interface or did I?”

The fundamental reason why tenants cannot get answers for these questions is the way today’s

IaaS cloud platforms present an abstract view of the states of resources to tenants. IaaS cloud platforms simplify the work of cloud tenants by providing clean virtual resource abstractions. Specifically, IaaS cloud platforms maintain the states of resources in a central database and forward the information to tenants when tenants query the platform. The problem is that the states recorded in the database are not necessarily identical to the real-world states of resources. We call this notion ‘functional (in)consistency’ of a resource:

**Def.** If a virtual resource $X$ is *functionally consistent* at time $t$, the data object representing the state of resource $X$ in the system must correspond to the real-world state of resource $X$ at time $t$.

A resource can be functionally inconsistent either because the actual process of creating/deleting/modifying the resource is not yet complete, or because of a bug or misconfiguration of the system. Regardless, for cloud tenants, there is no way to identify such inconsistency, and this leads to uncertainty with respect to the cause when problems occur. For a clear answer, tenants must depend on support from the provider, which is costly for both tenant and provider.

Explicitly telling the tenants about inconsistencies can help the tenants help themselves. In other words, if a tenant can distinguish the state of a resource written in the central database from its state directly monitored in the cloud infrastructure and/or its state according to a unit tester, the tenant can answer the questions listed above by themselves.

In this work, we introduce CloudSight, a transparency-as-a-service framework that maintains the history of states of cloud resources from various vantage points, and two applications on top of CloudSight: a functional consistency verifier and a time-traveling cloud debugger. CloudSight dynamically inserts monitoring functions into the target IaaS system to monitor state-changes of resources in different vantage points, stitches together state information of the same resource monitored in different places, and maintains the state change history in a graph database. CloudSight applications can trace resource state histories of interest using the Gremlin graph traversal language [78].

The aforementioned fact that IaaS software systems are under active development introduces a significant challenge in the system design of CloudSight: compatibility. For example, OpenStack, one of the market-leading IaaS software systems, releases a new major version update approximately once every six months. To keep invasive monitoring tools such as CloudSight compatible, it is

---

2The term *functional consistency* originates from the real-time system research of Audsley et al. in 1993 [11].
required to update the expert-level domain knowledge about the target system and to patch the tools accordingly. Likewise, to increase the coverage of monitoring, similar effort is required. Thus, if a tool is made by simply understanding a specific version of the target system and modifying the target system, the amount of time and effort to maintain compatibility becomes significant. We combine two key techniques to resolve this challenge. First, CloudSight uses key clustering to automate updating domain knowledge of the target system. Key clustering is adopted from the research on entity resolution [37], which is about extracting, matching, and resolving entities appearing in heterogeneous records. Second, dynamic code injection directly permits CloudSight to inject monitoring functions into the target system without modifying the target system’s source code. This enables CloudSight to be applied to a cloud platform regardless of the software version of the cloud system. In addition, this technique also allows cloud providers to dynamically adjust the monitoring level by turning on/off desired monitoring functions.

In summary, we make the following contributions:

• We show how the visibility problem of cloud tenants can be reduced to the visibility of the functional consistency of resources, and show how the CloudSight transparency-as-a-service framework effectively realizes this visibility.

• We adopt an entity resolution algorithm from the distributed systems domain, which efficiently minimizes the effort to maintain domain knowledge of the target cloud system in CloudSight.

• We present a working prototype implementation of the CloudSight framework in OpenStack, as well as two applications, a functional consistency verifier and a time-traveling cloud debugger.

• We demonstrate the utility of CloudSight by diagnosing real problems, one of which is a critical security bug first identified by our work [66].

2.2 Motivation

In this section, we motivate CloudSight with a case study involving the troubleshooting of security group bypassing problems. A cloud security group is a set of packet filtering rules that enables cloud users to realize custom firewall functionality. For example, in OpenStack, network communication is blocked by default and allowed only if it is enabled by a specific security group rule. Briefly, security group rule functionality is realized as follows: When a user requests an update to a security group rule, the cloud controller first updates the corresponding database entry, then publishes an RPC message, which is subscribed to by agents on the cloud compute nodes that realize the rules. At a
compute node, a security group rule is realized in the form of an iptables rule.

Security group bypassing means a situation when a packet without a corresponding security group rule reaches a VM. Packets may bypass security groups for various reasons. It can be because some rules, while deleted by the tenant, have not been deleted in the underlying cloud system (e.g., due to system delay or the failure of a software agent), or because old rules are being used for stateful caching [64]. Bypassing can also happen if a security group is disabled in the OpenStack configuration [68], or accidentally disabled because of an incorrect setting in the system configuration [66]. When cloud users confront a security group bypassing problem, the action they need to take varies depending on the cause. However, in practice, it is difficult to distinguish the cases from each other since their effects seem the same from the perspective of cloud users.

Interestingly, each listed root cause demonstrates functional inconsistency in a different way. Table 2.1 summarizes state changes of security group rules from different vantage points. In the case where the rule deletion is delayed, functional inconsistency happens temporarily between the cloud DB and iptables. However, if the inconsistency persists, it implies a failure in an internal OpenStack component (such as Neutron-OVS-agent in this case). For the stateful cache case, the inconsistency cannot be observed since the bypassing is observable only from the connections made before the rule was updated. If OpenStack itself is configured to disable security groups, the deleted rule was never created even though it appears in the database, so inconsistency occurs between the DB and iptables. Finally, if an external factor is preventing iptables from functioning, the inconsistency is observed between iptables and the unit test.

Suppose a cloud user confronted a security group bypass problem on his VM storing sensitive information. For security, the very first thing he may need to do is to pause the VM until the problem

Table 2.1: States of deleted security group rules. States of security group rule can be observed in three places. Cloud DB is the central database of cloud. iptables are the way OpenStack realizes security group rules, so actual creation/deletion/modification of rules can be observed by monitoring iptables entries. Since iptables may fail to filter packets, one can additionally monitor if each individual packet is hitting any iptables rules using a specialized unit test, cf. Section 2.4.

<table>
<thead>
<tr>
<th>Root Cause</th>
<th>Cloud DB</th>
<th>Iptables</th>
<th>Unit Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delay</td>
<td>Deleted</td>
<td>Exists → Deleted</td>
<td>Exists → Deleted</td>
</tr>
<tr>
<td>Agent Failure</td>
<td>Deleted</td>
<td>Exists</td>
<td>Exists</td>
</tr>
<tr>
<td>Stateful Cache [64]</td>
<td>Deleted</td>
<td>Deleted</td>
<td>Deleted</td>
</tr>
<tr>
<td>OpenStack Config. [68]</td>
<td>Deleted</td>
<td>Never Existed</td>
<td>Never Existed</td>
</tr>
<tr>
<td>External Factor [66]</td>
<td>Deleted</td>
<td>Deleted</td>
<td>Never Existed</td>
</tr>
</tbody>
</table>
is fixed by the cloud provider, which can be costly if it takes a long time. Further, the guaranteed
response time of cloud providers is never short (e.g., 12 hours with a $29 monthly fee [9]). Moreover,
if the problem was not from the provider’s side (e.g., Stateful Cache case), the cost will be a waste.
With CloudSight, the cloud user can instead self-diagnose and identify which manifestation of the
problem he is experiencing. Also, in case the problem is associated with the provider, he can quickly
report the diagnosed result by referencing the appropriate cloud element involved, so that the provider
also can readily understand the problem and take appropriate action.

2.3 CloudSight Architecture

The CloudSight architecture in the context of an IaaS cloud platform is shown in Figure 2.1. The components of a generic IaaS architecture is shown at the top. As shown, CloudSight inserts
loggers into existing cloud components to obtain comprehensive monitoring of the infrastructure.
The logs obtained by the loggers are stored in log storage and form the basis for CloudSight’s
transparency-as-a-service abstraction. Specifically, the logs are processed to generate a resource

Figure 2.1: CloudSight architecture.
graph, which essentially represents changes of resources’ states in both the cloud database and the cloud infrastructure. The information contained in the resource graph can be queried via the CloudSight API by tenants and tenant applications.

Figure 2.2 depicts the operational phases of CloudSight. A variety of cloud data sources are collected from the cloud platform and transformed into graph format to generate a log graph. The log graph forms the main input for the offline learning phase. Specifically, we apply semantics-based clustering, termed key clustering, to associate related data from different data sources. The cloud operator applies domain knowledge to the output of this step to generate a resource processing schema. During the operational phase this schema serves as input to generate a resource graph, which represents the means whereby CloudSight exposes information to cloud tenants. In Section 2.3.1 and Section 2.3.2, we first describe how CloudSight collects logs and generates the resource graph in the operational phase. In Section 2.3.3, we describe key clustering, which is the core of the learning phase of CloudSight.

### 2.3.1 Instrumentation and Log-Preprocessing

To obtain data from the IaaS platform, CloudSight uses a series of event Loggers to instrument the cloud system (Figure 2.1). A CloudSight event log is a nested dictionary structure (i.e., nested attribute-value pairs). The required attributes for each event are timestamp, log_type, resource_type, and method. The log_type attribute defines which vantage point the log was collected from (e.g., metadata if cloud DB), resource_type specifies the main resource the event is dealing with (e.g., virtual_machine, network or port), and method is a unique
identifier for each vantage point, which is usually the function name. Beyond this basic information, CloudSight loggers require minimal domain knowledge about the cloud platform. Specifically, because CloudSight key clustering associates information from different data sources, a logger developer can simply dump data into an event log, without attempting to interpret the data in the broader cloud context. We describe the implementation of loggers in OpenStack in Section 2.4.

CloudSight first preprocesses the collected event logs to narrow down the range of semantics of each attribute. To be specific, each nested attribute-value structured log is flattened into a one-level attribute-value structure by concatenating the chain of attributes of each innermost value. We call this chain of attributes a combinational key. For example in Figure 2.3(b), the edge labels others.networks.uuid, security.groups, method, and so on are all combinational keys, derived from the nested log shown in Figure 2.3(a). A combinational key can be understood as a contextualized attribute. For example, the combinational key others.networks.uuid in Figure 2.3(b) can be distinguished from another combinational key, e.g., instance.nova.object.data.uuid, which cannot be distinguished if not flattened, since both have the same last attributes uuid. The preprocessed logs are stored in the graph database, so we call the graph storing flattened logs a log graph.

### 2.3.2 CloudSight Resource Graph

CloudSight follows a resource-centric approach in presenting cloud information to cloud tenants. Data from different cloud sources are combined in a resource graph. This represents a natural way for tenants to think about their cloud resources and the relationships between them.

Figure 2.4 illustrates how different cloud resources are associated in a graph structure. A VM created by a tenant via the cloud API becomes a node vertex in the graph (VM1) with a set of properties (i.e., its ID, Name, the fact that it has been Created and its Type. ‘meta’ indicates that this is a cloud database object associated with a VM). In this example, the cloud controller decides to instantiate the VM on a physical machine (PM1), resulting in an assignedHost relationship between the two vertices in the graph. Similarly, once the hypervisor on host PM1 instantiates the VM, a Type : VM node (vm001) is created in the graph with an isHosting and instantiatedIn relationships with vertices PM1 and VM1. Note that with reference to the data sources identified in Figure 2.1, the data for vertices VM1, PM1 and vm001 comes from the API server, the Cloud DB, and the distributed components, respectively.
Figure 2.3: Log-to-graph conversion. (a) A nested attribute-value structured log is transformed into (b) a single level attribute-value structured log. This preprocessed log can be converted into (d) state information of resources at a specific time in the resource graph by referencing (c) the resource processing schema.

Figure 2.4: Capturing relationships between cloud resources.
To capture the history and state transitions associated with cloud resources, CloudSight employs list-based property cardinality (allowing multiple properties with the same property key), as well as nested properties (allowing a property to be contained in another property). We illustrate this functionality with the example shown in Figure 2.5. The figure shows a series of events (in the table in Figure 2.5) as well as the effect those events have on a (partial) resource graph. The series of events concerns a tenant request to create a VM (1), the VM creation being recorded in the cloud database and a specific physical machine selected to host the VM (2) and the VM being instantiated on the host (3). This is followed by the reciprocal set of events (4 to 6) when a request to delete the VM is executed. Figure 2.5(a) shows the effect on the graph based on the tenant’s create request (obtained from the API server). The \textit{Created} : \textit{T} property is associated with the \textit{VM1} vertex and the nested property functionality is used to record the \textit{EventLogID} (\textit{LID}) that triggered its creation. Figure 2.5(b) shows the result of the database log for the create request (event 2). Specifically, the \textit{VM1} vertex now has two \textit{Created} : \textit{T} properties, each with different nested \textit{LID} properties.

<table>
<thead>
<tr>
<th>Event Log ID (LID)</th>
<th>Data Source</th>
<th>Event Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Cloud API</td>
<td>Create VM1</td>
</tr>
<tr>
<td>2</td>
<td>Cloud DB</td>
<td>VM1 creation confirmed, hosted on PM1</td>
</tr>
<tr>
<td>3</td>
<td>Hypervisor API</td>
<td>VM instance vm001 created on PM1, with VMID for VM1</td>
</tr>
<tr>
<td>4</td>
<td>Cloud API</td>
<td>Delete VM1</td>
</tr>
<tr>
<td>5</td>
<td>Cloud DB</td>
<td>VM1 deletion confirmed.</td>
</tr>
<tr>
<td>6</td>
<td>Hypervisor API</td>
<td>VM instance #001 deleted</td>
</tr>
</tbody>
</table>

\textbf{Figure 2.5: Resource graph built by a series of events.}
Figure 2.5(c) shows the graph after the VM has been instantiated. Finally, Figure 2.5(d) shows the state of the graph once the series of delete events has executed.

Since the functional consistency of a cloud resource may vary as time progresses, cloud troubleshooting often involves investigating states of resources at a specific time (or within a specific known time range). The resource graph can easily be projected onto a time-plane to facilitate such efforts. Specifically, we can create a projected time-plane subgraph by selecting vertices whose event log timestamps fall within the time range of interest.

The CloudSight resource graph maintains a complete history of the cloud resources and their state transitions over time. Allowing cloud tenants direct access to the complete graph is inappropriate as it contains information for all tenants, as well as information the cloud provider might consider proprietary. Similar to time-plane projection, the CloudSight resource graph can readily be projected onto a tenant-plane. The base approach to project the resource graph onto a tenant plane involves creating a subgraph limited to vertices that have direct edges to the tenant vertex. We can then extend the subgraph to include every connected edge and neighbor vertex and can obfuscate or hide properties associated with neighbor vertices based on the cloud provider’s domain knowledge (applied during the learning phase).

2.3.3 Key Clustering

Event sequencing approaches based on metadata propagation are generally invasive as they require changes to the system itself [35, 48, 55]. Schema-based approaches are generally less invasive, but require manual construction of schema to associate events from different data sources [17, 41]. Thus, both approaches needs extensive domain knowledge, and the need is exacerbated in cloud platforms. This is especially true for open source efforts. Not only are these platforms under active development but they typically also have a large number of contributors with relatively loose coordination between them. Since different developers may use different attribute names for the same entity in different components, understanding the entire system through source code to update domain knowledge is not an easy task. For example, in OpenStack, in different functions and messages, the argument names device_id and routers are used to refer to the UUID of a virtual router, but device is used to refer to the UUID of a virtual interface; also, uuid is used to refer to the UUID of many entities including virtual routers and virtual interfaces.

With CloudSight event sequencing, we essentially follow a schema-based approach, but reduce
the domain knowledge required by automatically updating the event schema during the learning phase through key clustering, which is inspired by Entity Resolution research [37]. Key clustering is based on the following syllogism: if we know 1) attribute \( \text{Attr}_1 \) denotes \( X \), and we know 2) attribute \( \text{Attr}_2 \) is the same as attribute \( \text{Attr}_1 \), then we can conclude 3) attribute \( \text{Attr}_2 \) denotes \( X \). Here, the key challenge is how to obtain proposition 2), i.e., semantic relations between attributes. CloudSight gathers the semantic relations of attributes by clustering attributes based on the following semantic definitions. We define the Value Space of an attribute \( \text{Attr}_x \) as the set of values that attribute \( \text{Attr}_x \) ever had in the given learning data set, and denote it as \( V(\text{Attr}_x) \). Then, the relationships between attributes are:

\[
\begin{align*}
\text{Attr}_1 \text{ is a synonym of } \text{Attr}_2 & \iff V(\text{Attr}_1) = V(\text{Attr}_2) \\
\text{Attr}_1 \text{ is a hyponym of } \text{Attr}_2 & \iff V(\text{Attr}_1) \subseteq V(\text{Attr}_2) \\
\text{Attr}_1 \text{ is a hypernym of } \text{Attr}_2 & \iff V(\text{Attr}_1) \supseteq V(\text{Attr}_2)
\end{align*}
\]

where a hyponym (hypernym) means a more specific (generic) term. Based on this definition, we can cluster entire attributes by grouping synonyms in a cluster, namely a Synonym Cluster. Moreover, we can define the hierarchy among clusters by making hyponym clusters and hypernym clusters to point to each other. As a consequence, if we know the meaning of an attribute, we can infer the meaning of the other attributes in the same cluster as well as the generic meaning of attributes in all hyponym clusters.

A limitation of this approach is that when two key clusters have some common elements (i.e., they are co-hyponyms), but have no hypernym cluster, we cannot infer the relationship between the two clusters. Thus, we additionally define an artificial cluster, named Master Cluster, which become a hypernym cluster of a set of key clusters that have any common value. Master clusters can be built by first defining a master cluster for every key cluster and merging two master clusters in case their hyponym clusters have any element in common. Algorithm 1 shows the complete steps for CloudSight key clustering. Here, \( A \) refers to the set of every attribute (i.e., combinational key), which has property values (value space). \( \text{Combinations} \) refers to a function returning all of the N-element subsets a given set, and \( \text{Merge}() \) refers to a function that merges two master clusters and updates their sub-clusters accordingly.

Naively applying this approach presents a practical problem in that there exist nearly universally referenced values such as 0, 1, null, and None. Through an empirical study, we have listed values
Algorithm 1 Key Clustering

1: C ← φ
2: for each attr ∈ A do
3:   hasCluster ← False
4:   for each c ∈ C do
5:     if c.values = attr.values then
6:       c.keys ← c.keys ∪ {attr}
7:       hasCluster ← True
8:         break
9:   if hasCluster = False then
10:      c ← new Cluster()
11:      c.values ← attr.values
12:      c.keys ← {attr}
13:      c.super ← φ
14:      c.sub ← φ
15:      C ← C ∪ {c}
16: for each c₁, c₂ ∈ Comb(C, 2) do
17:   if c₁.values ∈ c₂.values then
18:      c₁.super ← c₁.super ∪ {c₂}
19:      c₂.sub ← c₂.sub ∪ {c₁}
20:   else if c₂.values ∈ c₁.values then
21:      c₂.super ← c₂.super ∪ {c₁}
22:      c₁.sub ← c₁.sub ∪ {c₂}
23: M ← φ
24: for each c ∈ C do
25:   if c.super = φ then
26:      M ← new Cluster()
27:      M.values ← c.values
28:      M.sub ← c.sub ∪ {c}
29:      c.super ← {M}
30:     for each c₁sub ∈ c.sub do
31:      c₁sub.super ← c₁sub.super ∪ {M}
32:      M ← M ∪ {M}
33: for each M₁, M₂ ∈ Comb(M, 2) do
34:   if M₁.values ∩ M₂.values ≠ φ then
35:      Merge(M₁, M₂)
36: return (C, M)

that are referenced by too many clusters and removed them prior to clustering so that we can avoid a single master cluster pointing to all key clusters.

Once we finish key clustering, we can infer the meaning of combinational keys in a synonym cluster if we know the meaning of any combinational key in it. One of the most practical sources of initial domain knowledge is the database of the target cloud, where most of the existing keys are used and may be documented by developers. Since this metadata is included in the data used by CloudSight, this provides a reasonable starting point for providers to apply their domain knowledge. Figure 2.6 shows an example of a cluster obtained by our clustering algorithm. Since the cluster contains the vm.id metadata attribute, and since all the other attributes had the same value space, the provider can reasonably deduce that all attributes in the cluster mean an “identifier of a virtual
Figure 2.6: Synonym clusters to update the schema.

machine,” and update the resource processing schema accordingly. Similarly, any hyponym clusters can simply inherit the meaning of the hypernym clusters. For example, if we know vm means virtual machine, and running_vm is a hyponym of vm, then we can regard every value referred to by running_vm as a virtual machine. Thus, such clusters can still be used in case there is no known key in them.

2.4 Implementation

We now describe our prototype implementation of CloudSight in the OpenStack cloud computing environment.

2.4.1 Instrumentation

The CloudSight loggers are implemented based on OpenStack Icehouse with Neutron OVS Hybrid networking (ML2) in a minimally invasive way; OpenStack components are implemented in Python and CloudSight loggers “hook” the target OpenStack components without changing the original source by replacing the entry module for each component. The substitute is essentially the same as the original, but it has additional code that dynamically patches selected functions of OpenStack by importing them in advance and wrapping them with a CloudSight wrapper. Table 2.2 summarizes the list of the functions we hooked and what the injected code does in each function. In addition to these loggers, CloudSight also monitors RPC messages exchanged among OpenStack components to trace the state changes of resources at the edge.

2.4.2 Unit Test

Unit testing in CloudSight checks if the target resource is functioning by effectively mimicking controlled user behavior. This enables CloudSight to monitor the functional consistency of resources
Table 2.2: The OpenStack functions hooked by CloudSight.

<table>
<thead>
<tr>
<th>Function</th>
<th>Original Function</th>
<th>Injected Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>f1</td>
<td>The original function translates a security group rule into iptables rule. The injected code supports mapping from security group rules to iptables rules by inserting security group information into comment fields of iptables.</td>
<td></td>
</tr>
<tr>
<td>f2</td>
<td>The original function returns a network interface name by accessing dictionary object. The injected code changes a character of the returning value to support (security group) active logging.</td>
<td></td>
</tr>
<tr>
<td>f3</td>
<td>The original function updates iptables rules. The injected code monitors the changes of iptables, which show the real-world states of security group rules.</td>
<td></td>
</tr>
<tr>
<td>f4, f5</td>
<td>The original functions are LibVirt VM creation and deletion functions. The injected code monitors VM creation/deletion events at the edge, which show the real-world state changes of VMs.</td>
<td></td>
</tr>
<tr>
<td>f6, f7</td>
<td>The original functions are network resource API functions. The injected code monitors network API requests and their propagation to the cloud database.</td>
<td></td>
</tr>
<tr>
<td>f8, f9</td>
<td>The original functions are computing resource API functions. The injected code monitors computing API requests and their propagation to the cloud database.</td>
<td></td>
</tr>
</tbody>
</table>

in a more strict way: if a virtual resource X is not only existing, as it is supposed to, but is also functioning correctly. We have implemented example unit test functionality for security groups in the form of an active logger. Figure 2.7 depicts the security group active logger implementation. The goal of the active logger is to trace which security group rule is used for a specific packet. Once an active logging request is received, the active logger enables iptables rules tracing, sends a probe packet, and then disables tracing. The security group unit test can be realized by sending a probe packet directly into the bridge such that it passes through the tap devices. For ingress rules, the active logger can send a fabricated packet to qvo. The probe packet will go through qvb and qbr and reach the tap device. Since we cannot make a packet flow from the tap device to the bridge without aid

Figure 2.7: Security group active logger in OpenStack.
from the target virtual machine, realizing an egress probe packet is a bit more involved. We solve this problem by temporarily creating a fake interface that has the target tap device name as its prefix, and add a wildcard character at the end of iptables rules’ target device name so that the rule will also be applied on the fake device.

2.4.3 Platform-Independent Components

We used a combination of Logstash [33] and Elasticsearch [32] to implement the forwarding of logs from the cloud components to the central log storage system. We used Titan 1.0 as the graph database and Gremlin [78], a graph traversal language developed under the Apache TinkerPop project, as the frontend. The internal graph components such as the graph generator, graph projectors, and the key cluster learner are implemented in Gremlin-Groovy. The projected resource graphs can be delivered to tenants through a Gremlin Server. The Gremlin Server allows tenants to make queries using the Gremlin language or using applications written in the Gremlin language. The Time-traveling Cloud Debugger (Section 2.5.1) and the Functional Consistency Verifier (Section 2.5.2) are written in Gremlin-Groovy as example applications that interact with the Gremlin Server.

2.5 Applications

To demonstrate the efficacy of our framework, we developed two novel tenant-oriented applications that assist tenants in troubleshooting cloud problems.3

2.5.1 Time-Traveling Cloud Debugger

Debugging a large-scale distributed system, such as a cloud platform, often involves determining the state of the system at a specific point in time and tracking the changes over time. We developed a novel time-traveling interactive cloud debugger on top of CloudSight to assist tenants in tracking their resources and troubleshooting problems. With the cloud debugger, cloud users can set a timepoint, similar to setting a breakpoint in a typical software debugger. Once the timepoint is set, the tenant can query the CloudSight’s resource graph to probe the status of its virtual datacenter at that point of time. For instance, tenants can obtain a list of existing resources, the connections among the resources, and the properties of each of those resources at the timepoint. Moreover, tenants can also track changes in a desired resource as time progresses, or explore the entire history of the resource.

3A video, based on a demonstration of the CloudSight applications [15], is available from the project website: http://www.flux.utah.edu/project/tcloud.
Table 2.3 shows the commands that the cloud debugger currently supports.

An example use case for this CloudSight application involves investigating the OpenStack port disappearance bug ([OpenStack bug #1158684] [63]). In this case, a VM port (i.e., a virtual interface) silently disappears in certain versions of OpenStack. (Correct behavior involves deletion of ports implicitly created by OpenStack, while ports explicitly created by tenants are not deleted but only detached.) With the cloud debugger, the tenant could trace back to the moment when the port was deleted and identify the termination of the virtual machine associated with the port that triggered the port deletion [15].³

2.5.2 Functional Consistency Verifier

As discussed earlier, one of the main problems that tenants face in the cloud is to determine if their rented resources are functionally consistent. We developed a simple yet powerful application called Functional Consistency Verifier (FCV) built atop CloudSight to determine the functional consistency history of a given resource. We present a real-world case study below that demonstrates the verifier’s utility.

2.5.3 VM Co-residency Check

OpenStack supports an option called affinity server group (or anti-affinity server group) that tenants may use when spawning a set of VMs to place them on the same compute node. Tenants verify if the spawned VMs co-reside by retrieving the members of the specified affinity group. However, this information might be inaccurate and not reflect the accurate placement of VMs.

---

**Table 2.3**: Cloud debugger commands.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&gt; show tplist</code></td>
<td>list all available timepoints in a tenant space.</td>
</tr>
<tr>
<td><code>&gt; show reslist</code></td>
<td>list all resources belonging to a tenant.</td>
</tr>
<tr>
<td><code>&gt; set tp=(timepoint_index)</code></td>
<td>set the timepoint.</td>
</tr>
<tr>
<td><code>&gt; unset tp</code></td>
<td>unset the current timepoint.</td>
</tr>
<tr>
<td><code>&gt; next</code></td>
<td>move to the next timepoint.</td>
</tr>
<tr>
<td><code>&gt; show tp</code></td>
<td>show the current timepoint.</td>
</tr>
<tr>
<td><code>&gt; show (resource_index)</code></td>
<td>show the properties and connections of the resource at the current timepoint.</td>
</tr>
<tr>
<td><code>&gt; history (resource_index)</code></td>
<td>show entire historical changes of the resource.</td>
</tr>
<tr>
<td><code>&gt; dump</code></td>
<td>dump the snapshot of the virtual datacenter at the current timepoint into a graph.</td>
</tr>
</tbody>
</table>
example, a violation occurs when a VM in the affinity group is migrated. Since OpenStack only applies the affinity group parameter when a VM is spawned (not when it is migrated [62]), this breaks the affinity group and makes VMs non-coreident. Moreover, the affinity group is not updated after the migration process, preventing tenants from verifying the above-mentioned behavior.

With FCV, the tenants can trivially verify the co-residency by tracing the consistency history of the affinity group. An example output of FCV against an inconsistent affinity group is shown below:

```<Affinity Group (Affinity-Group-ID)>  
... (previous consistency states) ...  
[From time-xxx to now] inconsistent  
(PM associated, multiple PM exists)```

### 2.5.4 VM Unreachable

An OpenStack security group consists of a set of rules that enables tenants to specify the type and direction of traffic accepted by VMs.

Security group functionality is relatively complex, and therefore it is easy for tenants to misconfigure it. Misconfiguration can also occur on the provider side either at the enforcement or during the translation into iptables rules. Again it is difficult for tenants to distinguish between two cases. A frequent problem faced by tenants is related to VM unreachability. After spawning a VM, tenants fail to login into the VM even when the connection on the SSH port is permitted via the security group rule. While this might simply be a tenant error, the problem might be related to a provider issue. For example, the problem might be due to a failed Neutron-OVS-agent. If the agent is down, even though Neutron updates the security group metadata according to the tenant’s request, the update would not result into iptables rules. Under this condition, if a tenant asks the cloud controller if the security group rules are applied to the VM, the cloud controller will retrieve the metadata and (incorrectly) inform the tenant that the requested security group rules are applied.

A cloud security group is a set of packet filtering rules that enables cloud users to realize custom firewall functionality. This functionality is somewhat less understood, and therefore easy for novice tenants to misconfigure security group rules. Security group malfunction can also be the result of errors on the cloud provider side. Without CloudSight functionality, it is nearly impossible for users to distinguish between these cases. Below we show, with two practical scenarios, how CloudSight can help users to diagnose security group problems more clearly.
Neutron (the OpenStack network resource manager) uses iptables to enforce security group rules, and the Neutron-OVS-agent is the component responsible for updating the iptables. Therefore, if the agent is down, even though Neutron updates the security group metadata according to the user’s request, the update would not be reflected in the iptables rules. Moreover, because security group rule updates are delivered via a message publish/subscribe interface, the cloud controller cannot readily determine that the Neutron-OVS-agent is down. Under this condition, if a user asks the cloud controller what security group rules are applied to the VM, the cloud controller will retrieve the metadata, and (incorrectly) inform the user that the requested security group rules are applied.

How can CloudSight diagnose the problem? Let’s say a user requested to add security group rules A and B to VM1, but while applying the rules, the Neutron-OVS-agent running on the host went down, so it applied rule A but not B. The user can use FCV to diagnose this; FCV automatically checks the history of consistency; the user simply needs to specify the security group rule ID.

CloudSight diagnoses the problem effectively. Let us assume a tenant requests to add two security group rules A (accept on port 80) and B (accept on port 22) to VM1. After installing rule A, but before installing the rule B, the Neutron-OVS-agent running on the host crashes. Since rule B is not enforced, the tenant cannot login into its VM. On noticing the problem, the tenant invokes FCV for both resources — A and B, which will show the inconsistency of rule B. The following output shows the example result of FCV against an unreachable VM:

```
<SGR rule-A-ID for VM VM1(VM1-ID)>
... (previous consistency states) ...
[From time-xxx to time-yyy] inconsistent
    (port associated, SGR exists)
[From time-yyy to now] consistent (IPTR created)

<SGR rule-B-ID for VM VM1(VM1-ID)>
... (previous consistency states) ...
[From time-zzz to now] inconsistent
    (port associated, SGR exists)
```

By comparing the result from rule A, which is consistent, the tenant learns that the creating IPTR (i.e., IPTable Rule) step is not completed for rule B.
2.5.5 Firewall Rules Not Enforced

As introduced in Section 2.2, even when all the OpenStack components are operational, a security group rule may not operate as expected if the host network environment is not configured correctly. One such scenario is when the network configuration variable `net.bridge.bridge-nf-call-iptables` is set to 0 [66]. This variable decides whether the packets traversing through Linux bridges should be filtered by iptables or not.

With the verifier application we enable tenants to identify such problems via CloudSight’s security group unit tester. Let us assume that a tenant adds a rule A (allow connection on port 443) and deletes another existing rule B (allow connection on port 80) for VM1. The variable `net.bridge.bridge-nf-call-iptables` is mistakenly set to 0 all along. When the tenant discovers that the rule B still appears to be applied, he first employs the verifier application to probe if rule B deletion occurred successfully or not. Then, FCV will show a result as follows:

```
<SGR rule-B-ID for VM VM1(VM1-ID)>
... (previous consistency states) ...
1: [time-xxx to time-yyy] inconsistent (SGR deleted, IPTR exists)
2: [time-yyy to now] consistent (SGR deleted)
```

where line 1 shows temporary inconsistency due to system’s delay from the database update to the actual iptable rule deletion, and line 2 indicates that rule B is now ‘consistently deleted.’ On discovering that the rule is in a consistent state, to further investigate the problem, the tenant turns towards the CloudSight unit test service. The first step is to craft a packet matching the specification of rule B, i.e., TCP port 80 from 10.x.x.x to VM1.

```
POST /api/activelog/security_group_probe
parameters : {’vm id’: (VM1’s id),
              ’direction’: ’ingress’, ’protocol’: ’tcp’,
              ’ip’: ’10.x.x.x’, ’port’: ’80’, ...}
```

After fabricating a packet according to the specification, the unit tester injects it to the OVS side of virtual interface (see veth(qvo) in Figure 2.7). The packet flows to the paired virtual interface (qvb) and the tap device connected to VM1. Since iptables is not active on the tap device (because
the configuration variable is set to 0), the packet fails to hit any of the iptables rules and reaches VM1. A further test against the rule A shows that even packets matching rule A are not actually hitting the rule. Specifically, the verifier will indicate an inconsistency: an iptable rule was created, but not observed in the log generated by the unit tester. The tenant notices the following message for the rule A:

[at time-zzz] inconsistent (IPTR created, IPTR not observed)

This reinforces the tenant’s suspicion that the problem is related to the iptables enforcement and informs the provider along with the related the resource graph elements.

2.6 Evaluation

We evaluated CloudSight in five ways: overall accuracy, robustness of key clustering, instrumentation overhead, log storage scalability, and graph processing latency. For the cloud infrastructure, we used a topology consisting of a controller node, a network node, and three compute nodes. For each node, an Emulab [93] d710 node [87] was used.

2.6.1 Key Clustering Robustness

Because CloudSight does semantic clustering, we do not rely on unique identifiers for entities. In other words, even if two entities (resources and properties) are represented by the same value in logs (e.g., \texttt{vmid:3333 and network\_ID:3333}), if their attributes are different and the attributes belong to different synonym clusters, CloudSight will correctly classify them as different entities. If, however, the value spaces (Section 2.3.3) of two different attributes completely overlap, the clustering process (learning phase) might misclassify the attributes into the wrong cluster. (The case of completely overlapping value spaces is depicted in Figure 2.8c.) We evaluated the robustness of CloudSight’s clustering process against such overlapping value spaces by artificially increasing the overlap in value spaces in our data. The data set for this experiment consists of 40,445 event logs with 896 different attributes, 69,902 different values, and 1,119,984 attribute-value pairs. For this evaluation we gradually change the overlapping attribute values in our data set in each iteration. Specifically, in each iteration we randomly picked 2,000 different values from event logs and changed one to another, which results in decreasing the total number of values by 1,000. For example, if value 3332 is picked from Figure 2.8a and changed to 3333, this modification can introduce a partial overlap of IDs for virtual machine and network (Figure 2.8b). Likewise, if all the values in the
value space of network IDs \{3330,3331,3332\} are chosen and changed to the values in the value space of virtual machine ID \{1111,2222,3333\}, it will introduce a complete overlap, and eventually incorrectly merges synonym clusters. Note that this process also can decrease the sizes of value spaces by choosing a value and changing to another in the same space. As a synonym cluster is incorrectly merged to another, the total number of synonym cluster decreases by one. Thus the reduced number of synonym clusters \(n\) can indicate the range of the number of incorrectly merged synonym clusters \((n + 1\) to \(2n\)).

Another accuracy factor that changes as the values overlap is incorrectly merged Master clusters (Figure 2.8b). Since key clustering correlates any possibly related synonym clusters through master cluster, heterogeneous clusters can be correlated through incorrectly merged master clusters even if they have any single values overlapped. Like the synonym cluster, the number of incorrectly merged master clusters can be estimated by counting the reduced number of master clusters.

Figure 2.9a shows the results. We observe that key clustering is remarkably robust at distinguishing synonyms (bottom line in Figure 2.9a); only when we reduced the number of values from 69,902 to 1,902, it incorrectly merged two synonym clusters. (This is the nearly imperceivable increase at the very right of the line.) In contrast, the number of incorrectly merged master clusters rapidly increased as soon as the values get overlapped. However, the impact of incorrect master cluster itself has modest impact since the correlation is used for helping cloud providers to infer unknown clusters.
2.6.2 Resource Graph Accuracy

The CloudSight process involves “reconstructing” resource graphs based on event logs from different data sources in the cloud platform. A basic metric of CloudSight accuracy is therefore whether the resulting resource graphs represent the “ground truth” user level request input. To evaluate this accuracy, we generated a synthetic workload mimicking multiple tenants and compared the resulting graphs with the requests by individual tenants. The workload consists of 100 different tenants, each of which concurrently requests creation and deletion of a random number of virtual machines, security groups rules, virtual networks, and virtual routers within a quota (i.e., 5 virtual machines, 80 security group rules, 10 networks and 10 routers per tenant). For the resource processing schema, we first obtained 185 synonym clusters through a learning phase and converted 92 of them consisting of 282 of combinational keys to the schema. Figure 2.9b shows the results.

In short, the resource graph completely covered every single resource requested, as well as all through possibly related related clusters.
implicitly created resources. Also, all the resources obtained the deleted: true property, which showed that the complete state transition was successfully reflected. The only missing record was an iptables rule for a default security group rule based on IPv6, which our current prototype does not support.

### 2.6.3 CloudSight Overhead

We evaluated the overhead of CloudSight by measuring the total run time and the CPU time of the OpenStack functions hooked by CloudSight compared to the same functions in an unmodified OpenStack. We only consider the loggers that inject code into the OpenStack original source. For the workload, we repeatedly created a VM and a security group rules and deleted them.

The routine internally calls a set of OpenStack functions hooked by CloudSight loggers and its logging agents. The details of the functions and injected code are shown in Table 2.2 in Section 2.4. Figure 2.9c shows the result. Except for functions f3, f6, and f7, we could see no significant overhead. Functions f6 and f7 have nontrivial overhead because the injected loggers perform additional I/O operations to read the metadata from the cloud database. For function f3, the overhead was relatively large because it executes an external command `iptables-save`. We have refactored these loggers to postpone the overhead-introducing operations using Eventlet [69], which resulted in nearly no direct overhead.

### 2.6.4 Scalability

The size of a log is proportional to the number of related events. However, some logs can increase more rapidly, especially as the number of related virtual machines increases. We have measured the change of log sizes by running a modified routine from the previous test; instead of creating and deleting single virtual machine, we created and deleted \( n \) virtual machines, where \( n \in \{5, 10, 15, 20, 25, 30\} \). We associated each virtual machine with 10 predefined security group rules and add one more rule after the virtual machine is created; i.e., each virtual machine creation will populate 11 iptable rules. The results are shown in Figure 2.9d.

We can notice the numbers of newly generated logs are roughly the same as the number of requests except for iptables logs and message logs. For iptables logs, the logs are multiplied rapidly because the number of logs is proportional to both the number of the created/deleted VM and the number of associated security group rules for each VM. In addition, when a new security group rule is added, some iptables rules may be reordered at the edge; since such reordering is also treated
as an update, the reordered rules are relogged. The rule order is used only for the security group unit testing. Therefore, by selectively turning on the relogging only if a security group unit test is requested, we could reduce the iptables logs to the same as the number of iptables rules.

2.6.5 Graph Processing Latency

We measured graph generation time as the number of logs to process increases. For each measurement, each graph was generated from scratch so the time is including the initialization overhead. Figure 2.9e plots the result. Excluding the initialization overhead, we observe the resource graph update takes 20 ms and log graph update takes 2 ms on average for each log.

2.7 Related Work

Our work relates to earlier efforts that can be classified as those aimed at assisting tenants in debugging and troubleshooting tenant-visible resources, i.e., tenant domain tools, and those that assist cloud providers to troubleshoot the cloud, i.e., provider domain tools.

2.7.1 Tenant Domain Tools

CloudWatch [8] and Ceilometer [65] provide visibility into the performance metrics of virtual instances. Sharma et al. [83] developed a tenant level end-point based cloud monitoring tool that allows tenants to conduct customized monitoring on their resources. Wu et al. [94] presented a cloud monitoring framework especially focusing on tenant network packet tracking. These works differ from CloudSight in that they focus on offering data-plane-only information, while CloudSight aims to provide visibility across both control and data plane. Amazon AWS CloudTrails [7] offers users an API history to track their operations in a virtual datacenter, similar to CloudSight’s logging requests from API servers. Jin et al. [47] presented a security group analysis tool based on security group metadata and sampled packets from physical switches. The scope of these cloud-level tools is typically limited to identifying symptoms of problems in their cloud resource instances. Our work is perhaps most closely related to the efforts of Wencheng et al. [92], who tried to resolve the visibility problem by offering a predefined-knowledge-based troubleshooting tool to cloud tenants. In contrast to these earlier efforts, CloudSight adopts a holistic approach that combines information from various places in the cloud, providing unique visibility to cloud tenants and assisting cloud operators in applying their domain knowledge to a complex and evolving cloud platform.
2.7.2 Provider Domain Tools

Ju et al. [48] built an intrusive failure-injection framework for OpenStack. Their framework could trace internal task flows to narrow down the root cause of a given OpenStack error. For the same problem, Sharma et al. [82] developed an online analysis tool based on RPC and API messages. Regarding the consistency problem of cloud resources, Xu et al. [98] introduced network consistency checking based on comparison of metadata from the cloud controller and the state of actual network resources on edge nodes. Madi et al. [56] adopted graph structure similar to CloudSight to cloud security compliance auditing. Likewise, Xiang et al. [96] used a similar graph structure as a general knowledge base for debugging cloud infrastructure. Compared to these works, CloudSight is unique in that it extends the consistency problem to the functionality of resources by offering unit test against resources through active logging as well as enabling tenants to troubleshoot cloud issues in a holistic manner.

More broadly, our work relates to debugging and troubleshooting of distributed systems, and specifically to solutions to the problem of associating events from different components in a distributed system [1, 17, 35, 41, 55]. In the latter category, CloudSight relates to two common approaches: metadata propagation, where metadata is “injected” into the operational system to allow event association [35, 48, 55], and schema-based approaches, where domain knowledge of the system is used to define an event schema to enable event association across the target system [17, 41]. CloudSight is conceptually closer to schema-based approaches as the CloudSight operational phase makes use of a resource processing schema. However, CloudSight’s semi-automated learning phase limits the domain knowledge required to make use of the approach.

2.8 Conclusion

CloudSight is a framework that provides IaaS cloud tenants greater visibility into their virtual data centers. With CloudSight, tenants can understand cloud mechanisms better, debug their cloud applications more precisely, diagnose even provider-side problems, and interact with providers more efficiently. We illustrated the utility of CloudSight by showing how it addresses real-world problems in the OpenStack cloud platform. Using CloudSight, we plan to explore a synergistic cloud paradigm where cloud providers and tenants can cooperate more efficiently to address cloud problems.
CHAPTER 3

PROVIDER-SIDE VISIBILITY INTO TENANT LEVEL NETWORK TRAFFIC

In this chapter, we address a network-level visibility problem from the cloud provider’s side to allow the cloud provider to obtain visibility of the load imposed by tenants.¹

3.1 Introduction

Network traffic accountability allows network operators to break down network usage and map it to consumers such as servers, virtual machines, virtual networks, or tenants. Fine-grained network accountability, such as determining the bandwidth of each point-to-point (i.e., host-to-host, or VM-to-VM) flow, is critical to ensure reliable cloud performance and customer satisfaction. For example, when a network hot spot appears, operators wish to determine which consumer utilizes the network more than it should, or which consumers may suffer performance degradation due to the hot spot.

Network accountability is challenging in multitenant datacenters, as today’s network systems cannot provide a line-rate flow measurement for datacenter-scale traffic. Researchers have proposed two classes of solutions to realize datacenter-scale network monitoring. Fine-grained flow sampling solutions such as sFlow [72] or NetFlow [24] can monitor individual IP flows at network devices, thereby simplifying network accountability. However, proposed datacenter-scale flow sampling relies on intrusive instrumentation either on network devices [34, 79, 80, 28] or on end hosts [20, 25]. Furthermore, sampling flows pervasively in a whole datacenter with high network coverage is prohibitively expensive, particularly at aggregation and core switches. More recent approaches make use of software-defined networking to enable flow-level counters that can account for bandwidth usage at switches [46]. However, these approaches also suffer from scalability concerns due to

¹This work is a minor revision of the paper published in ACM SoCC in 2017 (Baek et al., Polygravity: traffic usage accountability via coarse-grained measurements in multi-tenant data centers. In Proceedings of the 2017 Symposium on Cloud Computing, ACM. DOI: 10.1145/3127479.3129258) [13].
limitations in device memory and the number of flow rules that can be used for such measurements.

An alternative solution to reveal fine-grained flow usage relies on combining the network routing matrix with coarse-grained link-level measurements, e.g., SNMP. Initially developed for ISP networks, tomogravity [104] was also applied to datacenters [49, 44, 43], albeit with less accurate results (e.g., best-case average relative error around 15%). The main challenge for accurate traffic estimation with tomogravity in a datacenter is that datacenter traffic, naively interpreted, might not exhibit the inherent structure needed for this approach. Specifically, tomogravity was designed with two assumptions, which hold for ISP networks, about the network traffic characteristics: 1) all nodes proportionally contribute to overall traffic flows; and 2) the network does not have internal sinks/sources. Contrary to the first assumption, Kandula et al. [49] showed that the datacenter traffic matrix is sparse, and all nodes do not contribute to the traffic matrix. Contrary to the second assumption, datacenter networks may contain a variety of internal sinks and sources. For instance, due to software-defined firewalls [4, 67], host machines or SDN devices may be heavy internal traffic sinks. Likewise, due to broadcast-based image distribution [42, 50, 61] and port mirroring for Intrusion Detection or Real User Monitoring, the internal switches may act as heavy traffic sources.

Our key insights to overcome the challenge are as follows. First, datacenter administrators have access to readily available information about the contributing nodes, e.g., tenant-level virtual topology configuration or access control setups, which can be used to deal with fact that datacenter traffic flow is contributed by a limited number of nodes. Second, noise due to internal sinks/sources can be effectively canceled out by integrating information about their behavior into the tomogravity model. In this work, we show that utilizing such cloud configuration domain knowledge is key to precise traffic estimation. We propose a novel method called Polygravity, derived from the original tomogravity algorithm, to account for fine-grained flow usage in multitenant datacenters with heterogeneous domain knowledge. Polygravity performs significantly better than previous methods for datacenter network accountability. For tenants with fine-grained domain knowledge, Polygravity reduces the average relative error of estimating flow usage to less than 1%. For tenants with coarse-grained domain knowledge, with assistance of host-based partial sampling, Polygravity consistently reduces the relative error by $\frac{1}{3}$ compared to the relative error of the sampling-only solution.

To summarize, our contributions include the following.

- We identified domain knowledge integration as a key enabler to apply the gravity model to
the sparse traffic matrix estimation problem, and systematically adapted tomogravity to the variation of datacenter infrastructure, making Polygravity generally applicable to other datacenters (Section 3.3.2.2).

- We identified that the ‘no internal sinks/sources’ assumption of the tomogravity model does not hold in datacenter networks, and devised a Inner Gravity Estimation model for augmenting the domain of gravity models to internal sink/source nodes (Section 3.3.2.1).
- We designed the Polygravity model to selectively integrate additional estimation models such as sampling for better estimation, especially in case the given domain knowledge for a tenant is coarse-grained (Section 3.3.2.3).
- To thoroughly evaluate the performance of Polygravity, we generated realistic datacenter traffic by tenant-level traffic emulation based on previous datacenter traffic measurement studies [19, 49, 23] and simulating datacenters with different environmental setups (Section 3.4).

3.2 Background and Related Work

Formulating a traffic matrix to represent resource allocation is a general approach in network traffic engineering. The data sources used to construct such a traffic matrix often include link counts from SNMP data, routing information, topology information, and so forth. In this section, we first introduce how we construct a traffic matrix to formulate the traffic usage accountability problem, and then we explore the related work with respect to solving traffic matrices, and the applicability in datacenters.

3.2.1 Network Traffic Matrix Estimation

We formulate the problem of determining flow usage as follows:

$$\arg\min_t ||x - A \cdot t||$$  \hspace{1cm} (3.1)

where $t$ is the $n \times 1$ flow traffic vector$^2$ representing the traffic usage of each flow ($n$ denotes the number of flows), $A$ is then $m \times n$ routing path matrix revealing whether a flow traverses through each physical interface ($m$ denotes the total number of physical links), and $x$ is the $m \times 1$ link count vector.

Given that link counts $x$ and $A$ are commonly available, our goal is to determine the optimal

$^2$The $n \times 1$ traffic vector can be converted to $\sqrt{n} \times \sqrt{n}$ traffic matrix where $\sqrt{n}$ is the number of existing terminals. Thus, in this work, we use traffic matrix and traffic vector interchangeably.
solution $t = t_o$, which is the true traffic vector and naturally minimizes the term $||x - A \cdot t||$. A straightforward approach is using quadratic programming; e.g., applying the *Least Square Method* for Equation 3.1. However, in real-world datacenter networks, the number of flows is significantly larger than the number of links (i.e., $A$ is a fat matrix as $m \ll n$). There can be multiple solutions that satisfy Equation 3.1. Thus, a particular solution $t_p$ yielded by the least square is not necessarily the optimal solution $t_o$.

Network tomography refers to the methodologies that infer this optimal traffic matrix $t_o$ by using a limited number of measurements such as link counts $x$. Vardi [90] adopted a Poissonian model and employed an iterative approach that uses the EM algorithm [27] to find approximate solutions, which was the first to put the idea of network tomography into practice. Yu et al. [22] investigated the time-varying nature of the sender-receiver traffic by fitting the basic independent and identically distributed (i.i.d.) model locally using a moving data window.

### 3.2.2 Tomogravity Model

Given that there could be multiple solutions fitting Equation 3.1, it is intuitive to augment the traffic matrix with more external domain knowledge to constrain the search process for $t_p$ closer towards the optimal solution $t_o$.

In large-scale IP networks, Zhang et al. [104] solved the network traffic matrix estimation problem — *determining the traffic matrix representing the volume that flows from every ingress point into the network and to every egress point out of the network* — by proposing *tomogravity* modeling, which consists of *gravity modeling* and *quadratic programming* for refinement.

A typical gravity model views a target network as a black box surrounded by sites as shown in Figure 3.1, assuming that the network traffic from a site to another site is proportional to both the total traffic coming out from the source site and the total traffic coming into the destination site:

$$T(s, t) \propto T_{in}(s) \cdot T_{out}(t)$$

(3.2)

**Figure 3.1**: A view of gravity model to an example network.
where $T(s, t)$ denotes the traffic from site $s$ to site $t$, $T_{in}(s)$ denotes the total traffic going into the network from site $s$, and $T_{out}(t)$ denotes the total traffic going out from the network to site $t$.

Tomogravity uses gravity modeling as a way to obtain the initial traffic matrix model for an Autonomous System (AS). Zhang et al. suggested two variations of the gravity model: the Simple Gravity Model and the Generalized Gravity Model. The Simple Gravity Model estimates the traffic between two sites simply based on the network’s total incoming or outgoing traffic. Though the Simple Gravity Model is helpful to figure out the overall traffic exchanges, this model is based on an assumption that all the sites exchange traffic evenly, which can oversimplify traffic behaviors of real-world networks. The Generalized Gravity Model is proposed to overcome the even-traffic restriction. They classified the sites — “Access Link” and “Peer Link” — and customized the gravity model for each combination of source and destination type according to the routing policy.

Although the gravity model is excellent for capturing overall traffic patterns, the results may not be consistent with the interior network traffic, since it does not take interior network traffic into account, i.e., the link counts in the network blackbox in Figure 3.1. Accordingly, the tomogravity model performs a second-step refinement on the initial results $t_g$ obtained from gravity modeling by using the following quadratic programming:

$$\arg\min_{t_w} ||x_w - A \cdot t_w||$$

(3.3)

where $x_w = x - A \cdot t_g$, i.e., the link error of the initial model $t_g$.

Since a particular solution $t_{w,p}$ yielded by the Least Square method against Equation 3.3 has the smallest norm, the flow vector $t = t_g + t_{w,p}$ becomes: first, the solution closest to $t_g$ among the solutions minimize the objective function in Equation 3.1; second, the solution most consistent with the internal link counts on condition of Equation 3.1. After the second-step refinement, the final step of tomogravity modeling is replacing all negative values of $t_g + t_{w,p}$ with 0 and applying Iterative Proportional Fitting (IPF) [22] to minimize the link error.

### 3.2.3 Tomogravity Model in Datacenter Networks

The tomogravity model works well for large-scale IP networks but may not be applicable to modern datacenter networks. Kandula et al. investigated traffic characteristics in datacenters, and observed the datacenter traffic matrix is sparse in comparison with ISP networks. For this reason, the tomogravity model leads to a 60% median estimation error in their datacenter network. To improve accuracy, Hu et al. [44] proposed to utilizing additional information such as the ownership of virtual
machines (VMs) and shared jobs, achieving around 20% average relative error in the best case. Hu et al. [43] suggested to utilizing SDN switches to additionally measure aggregated flows. However, for a real-world deployment, it is prohibitive to exhaust available SDN rules only for improving traffic matrix estimation, considering the final improvement it achieved (around 15% best case average relative error). In contrast to these earlier efforts, Polygravity adopts domain knowledge of a target datacenter in a holistic way, allowing highly fine-grained and flexible customization of the estimation model. This makes Polygravity suitable for heterogeneous datacenter networks. In addition, our approach uniquely solves the internal sink/source problem, which enables Polygravity to tolerate heavy internal sink/source noise.

3.3 Methodology

As mentioned in Section 3.2, an intuitive attempt to determine traffic usage accountability in modern datacenter networks is to form a traffic matrix to represent flows and solve the matrix using tomogravity model. However, traffic properties in multitenant datacenters differ significantly from ISP networks in the following ways:

- There are tenant-isolation and function-virtualization factors in multitenant datacenter networks, e.g., VLAN, VXLAN, GRE tunneling, firewalls, etc. Such factors partition the network infrastructure and routing paths are accordingly segmented, which translates to a highly partitioned traffic matrix.
- Imbalanced incoming/outgoing traffic volume happens mainly due to various network-level applications such as software-defined firewalls, broadcast-based VM image delivery, hot-standby middleboxes, port mirroring for IDS, etc., leading to heavy “source” and “sink” spots [4, 67, 42, 50, 61].

We address these challenges by enhancing the tomogravity model through three phases: augmenting the domain of the gravity model with internal sinks/sources (in Section 3.3.2.1), redesigning the gravity model to flexibly reflect network dynamics (in Section 3.3.2.2), and allowing the integration of supplementary estimation models into gravity model (in Section 3.3.2.3).

3.3.1 Overview

We name the full stack of our methodology Polygravity (from its poly-morphic nature and utilizing the gravity model multiple times). Figure 3.2 summarizes the complete steps to conduct
Polygravity:

1. For a given data set within a time window, compute the ingress and egress traffic ($T_{in}$, $T_{out}$) of interior interfaces by conducting *Inner Gravity Estimation*. Then, augment the gravity model’s domain to include every interior interface that acts as a non-negligible sink or source. (Section 3.3.2.1).

2. Customize the gravity model to fit your datacenter network by classifying the sites and constructing a flow graph. Since this step is mainly about processing the metadata of the target datacenter, automation of this step is feasible through network management tools. (Section 3.3.2.2 Steps 1 and 2).

3. Apply the customized gravity model to the augmented gravity domain of the current data set and obtain the initial gravity model $t_g$ (Section 3.3.2.2 Step 3).

4. Conduct broadcast noise cancellation on the link matrix to minimize the impact of broadcast traffic (Section 3.3.3.2).

5. Apply the weighted Least Square Method, replace all negative values with zero, and apply Iterative Proportional Fitting (Section 3.3.3.1).

3.3.2 Constructing the Initial Traffic Matrix

In this phase of traffic matrix construction, we propose two approaches: the *Augmenting Gravity Domain* and the *Network Infrastructure Adaptation* in accordance with the challenges mentioned above.
3.3.2.1 Augmented Gravity Domain

Previous network tomography methods [90, 86, 22, 104, 44, 43] assumed the interior network sinks and sources are negligible. However, multitenant datacenters can seriously violate the assumption because these networks deploy various and complex techniques that make the interior network have heavy sinks (e.g., distributed firewall, virtual network host) or sources (e.g., multicast-based virtual machine image distribution, port mirroring for IDS). Consequently, the resulting traffic matrix elements are either overestimated or underestimated.

To expose the interior sources/sinks out of the black box, we perform a two-step approach: 1) estimating the amount of source/sink traffic of each interface of interior network devices$^3$ and 2) augmenting the classic sink/source domain to integrate the interfaces.

Though it is simple to determine if a network device is a source or a sink and by how much (by comparing the total amount of inbound and outbound traffic), an interface-wise assessment is never easy. Even if we know the amount of inbound traffic consumed by a (sink) device, we cannot simply decide how much was consumed by each individual interface. One may try computing the numbers by setting a system of linear equations and finding a solution that minimizes the error, but such a system is under-determined in most of cases so that it cannot give a single solution. If a device has $k$ interfaces, then there can be $k^2$ variables but only $2k$ equations.

As a way to model interior sink/source traffic, we applied the gravity model on each individual device and extracted the amount of sink/source traffic for each interface of the device, termed Inner Gravity Estimation (IGE). The underlying assumption of this approach is that the more traffic leaves (comes) through an interface and the less traffic comes (leaves) through the other interfaces, the more likely the interface is a source (sink).

Specifically, consider the following two simple gravity models:

$$U_I(l_i, l_j) = U_{in}(l_i) \cdot \frac{U_{out}(l_j)}{\sum_{l_k \in f(\gamma)} U_{out}(l_k)}$$  \hspace{1cm} (3.4)

$$U_O(l_i, l_j) = U_{out}(l_j) \cdot \frac{U_{in}(l_i)}{\sum_{l_k \in f(\gamma)} U_{in}(l_k)}$$  \hspace{1cm} (3.5)

where $l_i$ denotes an interface, $U_{in}(l_i)$ (or $U_{out}(l_i)$) denotes the amount of traffic that came into (or left

$^3$Here, an interior network device means a node that works as an intermediary device between the source and the destination. Thus, not only network routers and switches but also physical machines hosting VMs can be treated as interior network devices.
from) the device through the interface \( l_i \), and \( f(\gamma) \) means the set of all interfaces of an interior network device \( \gamma \). If we compute a gravity model for a network device only using Equation 3.4, because it divides every *inbound* traffic \( (U_{in}) \) proportionally, the resulting gravity model \( (U_I) \) preserves the consistency of the inbound traffic as in Figure 3.3(c). Likewise, the gravity model based on equation 3.5 \( (U_O) \) keeps the consistency of outbound traffic \( (U_{out}) \) as illustrated in Figure 3.3(d).

Let’s call each of these gravity models the *inbound* and *outbound inner gravity models*.

Then, IGE for each interior network device \( \gamma \) consists of the following steps:

1. \( \forall l_i \in f(\gamma) \), initialize \( T_{in}(l_i) = T_{out}(l_i) = 0 \).

2. Compute the total amount of incoming traffic \( U_{in}^{total} \) and outgoing traffic \( U_{out}^{total} \):

\[
U_{in}^{total} = \sum_{l_i \in f(\gamma)} U_{in}(l_i), \quad U_{out}^{total} = \sum_{l_i \in f(\gamma)} U_{out}(l_i).
\]

If \( U_{in}^{total} > U_{out}^{total} \), the device is a sink; otherwise, if \( U_{in}^{total} < U_{out}^{total} \), the device is a source.

3. If the device is a sink, compute outbound inner gravity models \( U_O \) for \( e \). Likewise, if the device is a source, compute the inbound inner gravity model \( U_I \).

4. If the device is a sink, for each \( l_i \in f(\gamma) \), compute the inconsistency of inbound traffic:

\[
U_{in}^{incon}(l_i) = U_{in}(l_i) - \sum_{l_k \in f(\gamma)} U_O(l_i, l_k)
\]

If \( U_{in}^{incon}(l_i) < 0 \), \( U_{in}^{incon}(l_i) = 0 \).

Likewise, if the device is a source, compute:

\[
U_{out}^{incon}(l_i) = U_{out}(l_i) - \sum_{l_k \in f(\gamma)} U_I(l_k, l_i)
\]

If \( U_{out}^{incon}(l_i) < 0 \), \( U_{out}^{incon}(l_i) = 0 \).

5. If the device is a sink, for each \( l_i \in f(\gamma) \), if \( U_{in}^{incon}(l_i) > 0 \), the interface \( l_i \) is likely a sink for the network. So, distribute the total amount of sink traffic proportionally as:

\[
T_{out}(l_i) = (U_{in}^{total} - U_{out}^{total}) \cdot \frac{U_{in}^{incon}(l_i)}{\sum_{l_k \in f(\gamma)} U_{in}^{incon}(l_k)}
\]

Likewise, if the device is a source, distribute the total amount of source traffic proportionally as:
Figure 3.3: Example Inner Gravity modeling for an interior network device acting as a sink. (b) shows the link counters for a device. Note that the device is acting as a sink because the total ingress traffic is larger than the total egress traffic; (c) shows the interface-to-interface traffic estimation based on the Inbound Inner Gravity and its resulting link traffic, which keeps the consistency of inbound link counters; likewise, (d) shows the results based on the Outbound Inner Gravity, that keeps the consistency of outbound link counters.

A more intuitive explanation of IGE is that, when an interface of a switch acts as a source, since the inbound inner gravity model on the switch cannot find further inbound traffic from other interfaces to supply the outbound traffic of the interface, it yields a traffic estimation that underutilizes the outbound traffic of the interface. Likewise, the outbound inner gravity model returns a traffic estimation that under-utilizes the inbound traffic of the interfaces acting as sinks.

Through conducting IGE over every interior network component, we can obtain a list of interior network sources and sinks as well as their approximate amount of traffic. Now, augment the domain...
of gravity model \( L \) (\( L \) is the set of all external nodes) to include these additional sources and sinks obtained through IGE:

\[
L_{\text{aug}} \leftarrow L \cup \{l_i | \max(T_{\text{in}}(l_i), T_{\text{out}}(l_i)) > \tau, \quad l_i \in f(\gamma), \forall \gamma \in \Gamma\}
\]

where \( \Gamma \) is the set of every interior network element, and \( \tau \) is a threshold of negligible sink/source size. We term this newly generated set \( L_{\text{aug}} \) the Augmented Gravity Domain. Note that, with the Augmented Gravity Domain, the total traffic that comes into and goes out from the blackbox network becomes equal.

### 3.3.2.2 Network Infrastructure Adaptation

Due to variation of network infrastructure and diversity of traffic patterns, without appropriately tailoring the gravity model, a direct application of the tomo-gravity model could yield poor estimation quality in reality [49]. Customizing the gravity model to fit a specific network is a process of reflecting network domain knowledge in the model, so a single model can hardly fit heterogeneous networks.

Through our experience in applying the gravity model to various datacenters, we have come up with a general approach for customizing the gravity model for different types of networks. In this section, we present the general approach with an example cloud shown in Figure 3.4(a).

![Example Cloud Network](image1)

![Augmented Gravity Domain](image2)

![Customized Gravity Model](image3)

**Figure 3.4**: An example walk-through of gravity model customization for a cloud network. (a) an example cloud network consists of one controller, two network gateway nodes, and three compute nodes hosting total of seven VMs. The left four VMs and the right three VMs belong to different virtual networks; (b) the result of gravity domain augmentation. The administrator assumed the compute nodes do not directly talk with VMs through the network so he excluded compute nodes’ (virtual) interfaces toward VMs from the augmentation; (c) a flow graph of a customized gravity model. The solid lines are unconditional edges, and the dotted lines are conditional edges. The numbers on the edges show the index of the set that each edge belongs to in the ordered set \( D \) of Algorithm 2.
As the first step, we classify the network sites in the gravity domain $L_{\text{aug}}$ (one can use the nonaugmented domain $L$ if interior sinks and sources are negligible). Classification of sites can simplify the process of cutting out unlikely existing flows from the gravity model in the next step. Let $C = \{c_1, c_2, \ldots, c_k\}$ be the set of all classes defined, so $c_1 \cup c_2 \cup \ldots \cup c_k = L_{\text{aug}}$ and $c_i \cap c_j = \emptyset$ if $i \neq j$. When classifying, the rule of thumb is that the more fine-grained the classification is, the sparser traffic matrix one may get. However, if no behavioral difference between two classes of sites is known, it is pointless to differentiate them. The six nodes in Figure 3.4(c) show an example classification of the network sites in the example datacenter.

The next step is generating a flow graph. We build a directed graph $G = (C, E)$ that describes the traffic relationships between classes. Here, we add edges according to our domain knowledge about the target network. An edge $e = (c_i, c_j)$ represents the existence of network flows from sites in class $c_i$ to sites in class $c_j$. An edge $e$ can be either unconditional or conditional. If unconditional, the edge indicates there exists a flow for every pair $(l_x, l_y)$, where $l_x \in c_i, l_y \in c_j$; if conditional, it means there exists a flow for some pairs. Such a condition can be defined using a condition function $h(l_x, l_y)$:

$$h(l_x, l_y) = \begin{cases} 1, & \text{if the flow } l_x \text{ to } l_y \text{ exists} \\ 0, & \text{otherwise (including } l_x = l_y) \end{cases}$$

The condition function always returns 0 for the flows that do not belong to any edges in the flow graph. For the unconditional edges, the condition function always returns 1 except $l_x = l_y$.

Figure 3.4(c) shows a flow graph for the cloud network in Figure 3.4(a). Here, the administrator may define several condition functions for the conditional edges. For example, since traffic between two VMs can exist only if the two belong to the same virtual network, the administrator can define the condition function as:

$$h(l_x, l_y) = \begin{cases} 1, & \text{if } l_x \text{ and } l_y \text{ belong to the same virtual network and } l_x \neq l_y \\ 0, & \text{otherwise} \end{cases}$$

where $c_{\text{vm}}$ denotes a class for VMs.

The last step is computing a gravity model. According to the flow graph $G$, we can systematically compute a custom gravity model as described in Algorithm 2. An intuitive explanation is that the algorithm iterates over each class and distributes either $T_{\text{in}}$ or $T_{\text{out}}$ to the connected edges. $\hat{T}$ appearing
Algorithm 2 Compute Customized Gravity

1: for each \( l_i \in L_{\text{aug}} \) do
2: \( T_{\text{in}}(l_i) \leftarrow T_{\text{in}}(l_i) - \sum_{k \in L_{\text{aug}}} \hat{T}(l_i, l_k) \)
3: \( T_{\text{out}}(l_i) \leftarrow T_{\text{out}}(l_i) - \sum_{k \in L_{\text{aug}}} \hat{T}(l_k, l_i) \)

4: \( E_{\text{used}} \leftarrow \phi \)
5: \( D \leftarrow \{(c, \text{dir}) \mid \text{dir} \in \{\text{in, out}\}, c \in C\} \)
6: Sort \( D \) by an arbitrary order
7: for each \((l_i, l_j) \in L_{\text{aug}} \times L_{\text{aug}}\) do
8: \( T(l_i, l_j) \leftarrow 0 \)
9: for each \( d = (c, \text{dir}) \in D \) do
10: \( E_{\text{temp}} \leftarrow \phi \)
11: if \( \text{dir} = \text{in} \) then
12: \( E_{\text{temp}} \leftarrow \{e \mid e \in E, e \not\in E_{\text{used}}, e = (c, c_x)\} \)
13: \( L_k \leftarrow \{n \mid \forall (c, c_y) \in E_{\text{used}}, n \in c_y\} \)
14: \( L_l \leftarrow \{n \mid \forall (c, c_z) \in E_{\text{temp}}, n \in c_z\} \)
15: for each \( e \in E_{\text{temp}} \) do
16: \( c_x \leftarrow e[1] \)
17: \( \forall l_j \in c, \forall l_j \in c_x, \)
18: \( T(l_i, l_j) \leftarrow \left(T_{\text{in}}(l_i) - \sum_{k \in L_k} T(l_i, l_k)\right) \cdot \frac{\sum_{l \in L_k}(T_{\text{in}}(l_i) \cdot \hat{T}(l_i, l))}{\sum_{l \in L_k}(T_{\text{in}}(l_i) \cdot \hat{T}(l_i, l))} \)
19: if \( T(l_i, l_j) < 0 \) then \( T(l_i, l_j) \leftarrow 0 \)
20: else if \( \text{dir} = \text{out} \) then
21: \( E_{\text{temp}} \leftarrow \{e \mid e \in E, e \not\in E_{\text{used}}, e = (c_x, c)\} \)
22: \( L_k \leftarrow \{n \mid \forall (c_y, c) \in E_{\text{used}}, n \in c_y\} \)
23: \( L_l \leftarrow \{n \mid \forall (c_z, c) \in E_{\text{temp}}, n \in c_z\} \)
24: for each \( e \in E_{\text{temp}} \) do
25: \( c_x \leftarrow e[0] \)
26: \( \forall l_i \in c_x, \forall l_j \in c, \)
27: \( T(l_i, l_j) \leftarrow \left(T_{\text{out}}(l_j) - \sum_{k \in L_k} T(l_k, l_j)\right) \cdot \frac{\sum_{l \in L_k}(T_{\text{out}}(l_j) \cdot \hat{T}(l_j, l))}{\sum_{l \in L_k}(T_{\text{out}}(l_j) \cdot \hat{T}(l_j, l))} \)
28: if \( T(l_i, l_j) < 0 \) then \( T(l_i, l_j) \leftarrow 0 \)
29: \( E_{\text{used}} \leftarrow E_{\text{used}} \cup E_{\text{temp}} \)
30: for each \((l_i, l_j) \in L_{\text{aug}} \times L_{\text{aug}}\) do
31: \( T(l_i, l_j) \leftarrow T(l_i, l_j) + \hat{T}(l_i, l_j) \)

in line 1–3 and 30–31 is a preestimated traffic model, which is explained in Section 3.3.2.3.

Since either line 18 or line 27 will be executed at most \(|L_{\text{aug}}|^2\) times and each line has \(O(|L_{\text{aug}}|)\) of complexity in the worst case, the worst-case complexity of the algorithm is \(O(|L_{\text{aug}}|^3)\). However, the actual computation is substantially smaller because the flows that do not belong to any edge will be set to 0 without going through the computation.

After the last step, we can obtain the gravity model for every network flow (i.e. \( T(l_i, l_j) \forall l_i \in L_{\text{aug}}, \forall l_j \in L_{\text{aug}} \)). We can express the resulting gravity model in a column vector as \( t_g = (t_1, t_2, \ldots, t_{n'})^T \) where \( n' = |L_{\text{aug}}|^2 \).
3.3.2.3 Reflecting Preestimation

The customized gravity model is based on the assumption that each network node accesses a limited number of peers, and such an access model is known to the datacenter operators. However, operators do not always have access to such information — e.g., cloud tenants may define a virtual topology in a very coarse-grained level, or a virtual topology itself is dynamically changing at the application level such as Apache Hadoop and Storm.

As a remedy to this practical challenge, Polygravity allows operators to utilize supplementary estimation models to make up the limitation of the gravity model. A random n-out-of-N sampling technique [105] works well for this purpose — the random packet sampling technique captures the heavy flows in a sparse traffic matrix well, but it unreliably estimates smaller size flows and generally performs poorly for dense traffic matrices, which exactly counter-balances the cons and pros of the gravity model. In addition, the n-out-of-N sampling technique is widely implemented as an industry standard such as sFlow [72] and NetFlow [24] and readily deployable in datacenters even without special hardware supports.

As described in Algorithm 2 lines 1–3 and 30–31, Polygravity takes preestimated traffic into account by first assigning the traffic to the preestimated model and then running the gravity model for the leftover traffic. Depending on the quality of the preestimation, however, the naive application of preestimated traffic may not help, or even harm, the quality of estimation. Thus, we suggest an adaptive application of preestimated traffic to Polygravity. Especially for sampling, we may utilize the coefficient of variation (CV) of a sampled flow as an approximate indicator of its accuracy. Since our purpose in utilizing sampling is sketching the overall portions of heavy flows in the sparse traffic matrix (which the gravity model cannot capture without domain knowledge), it is reasonable to preserve the proportion of each flow in the whole. Therefore, rather than adjusting each individual flow by its CV, we adjust the entire preestimated traffic according to the mean of CVs ($\bar{CV}$). To be specific, we regard $\bar{CV} < 0.5$ sufficiently precise to take the preestimation as a whole, and $\bar{CV} > 1.5$ overly imprecise to reflect, and proportionally decrease the portion of preestimation when $\bar{CV}$ is in between.

Compared to the case of utilizing sampling alone for estimating the traffic of an entire datacenter, sampling with Polygravity enables datacenter operators to selectively deploy sampling. This is a significant benefit that leads to saving resources from processing and storing samples throughout the datacenter.
3.3.3 Estimation Refinement

Although our customized gravity model captures the overall traffic pattern, it does not take the links in the black-box in Figure 3.1 into account. With only the gravity model, the resulting flow estimation might show high error rate for interior network links. In this section we describe how Polygravity refines the initial gravity model solution.

3.3.3.1 Quadratic Programming

The classic tomogravity method refines the initial gravity model result through the Least Square Method (LSM) and Iterative Proportional Fitting (IPF) as introduced in Section 3.2.2. The refinement step of Polygravity is basically identical to that of tomogravity. In other words, Polygravity first applies a weighted Least Square Method:

$$\arg\min_t \frac{||t - t_g||}{w}$$ (3.6)

where $t_g$ is the gravity model solution we got from Section 3.3.2, $w$ is a weighting factor and $\div$ is an element-wise division operator. Since $A$ is a fat matrix, there exist multiple solutions that satisfy $||A \cdot t - x|| = 0$, and the goal is finding a solution that is closest to the gravity model solution. Since the least square solution $t$ can contain negative values, we first change all the negative values to zeros and apply Iterative Proportional Fitting [22] until the link error reaches a certain threshold. For the weighting factor $w$, we followed the original tomogravity’s approach that uses the square root of the gravity model ($w = \sqrt{t_g}$).

3.3.3.2 Broadcast Noise Cancellation

One issue in using the Augmented Gravity Domain is that it increases the number of flows to consider in quadratic programming, which naturally increases computation time. It is reasonable to take such interior network flows into account if the flows are important and expected to exist (such as flows from compute nodes to Controller in Figure 3.4(c) or filtered traffic by firewall). However, if some flows are not worthwhile to refine at the expense of additional computation time, we can remove them from the refinement steps.

Interior flows introduced by broadcasting traffic can be considered as such “flows less worthwhile to refine”: in traffic matrix use cases such as hot spot detection, link-failure detection, resource allocation, and capacity planning, the main input is a point-to-point traffic matrix but not precise.
background noise traffic. In addition, the interior switch interfaces (i.e., the interfaces augmented due to broadcasting) introduce a significant number of flows compared with the other sites. A single interior switch interface can introduce nearly $|L|$ additional flows because the destination of flows from a switch can hardly be limited within some sites (e.g., the interior switch interface class in Figure 3.4(c) had edges to all the other classes in the flow graph, and the condition function for interior switch interfaces may merely check if the interface is toward the peer or not). Therefore, by removing the interior flows attributed to broadcast traffic, we may significantly reduce the computation cost. Let’s refer these flows as noise flows.

Since the noise flows actually exist, simply excluding them from the flow matrix $t$, will harm the accuracy of the estimation result. To minimize the impact of noise flows, we can use the initial gravity estimation result of noise flows. In other words, we can ‘cancel out’ the traffic possibly attributed by noise flows from the link matrix $x$ by computing:

$$x_{nc} = x - A_{(br)} \cdot t_{g(br)}$$  \hspace{1cm} (3.7)

where $A_{(br)}$ is a routing matrix of the noise flows, $t_{g(br)}$ is a flow vector for the noise flows, and $x_{nc}$ is the ‘noise-canceled’ link matrix.

### 3.4 Evaluation

To evaluate Polygravity, we derived a synthetic data set based on measurement results from earlier studies and then used this data to compare Polygravity with a number of earlier traffic estimation approaches. In Section 3.4.1, we first describe our methodology for realistic synthetic data generation. In Section 3.4.2, we use the generated data sets to validate Polygravity by comparing it with different earlier approaches. In Section 3.4.3, we show the impact of interior source/sink and validate IGE and the noise-cancellation technique. Finally, in Section 3.4.4, we evaluate the scalability of Polygravity by measuring the changes of accuracy and running time of Polygravity as the target datacenter scales up.

#### 3.4.1 Synthetic Data Generation

Our original target through this work is estimating tenant-level network traffic of public multi-tenant datacenters [75, 60]. For validation of our algorithm against these datacenters, we should have the complete sets of SNMP data $x$, routing matrices $A$ and flow-level data $t$ from the datacenters as well as the domain knowledge. Similar to the case of Zhang et al. [104], though we could
collect the SNMP data and the routing matrices from the target datacenters, the complete flow-level data collection was not available due to the privacy policy and the vendor implementation of flow collection. For one of the datacenters, we could obtain flow samples, but the maximum sampling rate was too low compared to the amount of datacenter traffic to apply the same technique as Zhang et al. [104] and to generate realistic data sets.

As a solution to this problem, we first generated sets of tenant-level traffic in the Emulab testbed [93] with a realistic setup based on previous measurement studies [19, 49, 23] and collected complete sets of flow data from them. We then synthesized the flow data into simulated multitenant datacenters, and computed a consistent SNMP data set from the routing matrix of the simulated datacenter and the synthesized flow data. In Section 3.4.1.1 and Section 3.4.1.2, we describe the detailed methodology to generate realistic tenant traffic for web-service-style tenants and map-reduce-style tenants, the most typical types of tenants in clouds. In Section 3.4.1.3, we describe how we synthesized the data and simulated datacenter traffic.

### 3.4.1.1 Web Service Tenant

A three-tier architecture [31] is a typical setup for web services. For a realistic web service tenant traffic generation, we first deployed a three-tier web application consisting of a front-end load balancer, 10 Wordpress middle-tier nodes, and a back-end MySQL database. In this setup, the load balancer receives requests from clients through a gateway and forwards each of them to a Wordpress node in a round-robin fashion. The Wordpress nodes interact with the back-end database to process the clients’ requests.

Next, we generated workload based on a real-world multitenant datacenter traffic pattern introduced by Benson et al. [19]. The authors observed the distributions of 1) number of active flows, 2) flow interarrival times, and 3) flow lengths in multitenant datacenters form lognormal distributions. Since these three variables are dependent, if we control any two to follow lognormal distributions, the other naturally forms a lognormal distribution. We picked the traffic pattern observed at a specific switch in the paper of Benson et al. [19] (named PRV23) where nearly 85% of observed traffic was HTTP (i.e., web service traffic) and implemented a workload generator that mimics web service clients by randomly generating flows with flow interarrival times and flow

---

4Benson et al. [19] and Kandula et al. [49] used the term *flow* to mean *a continuous sequence of packets* from a source to a destination, which differs from the term flow we have used — point-to-point flow. For clear distinction, we notate *flow* only if it means *a continuous sequence of packets having the same five tuples.*
lengths following lognormal distributions close to the patterns of the PRV$_2$. 

We ran the workload generator for around 70 minutes and obtained 14 traffic matrices at 5 minute intervals. Figure 3.5(a) visualizes the resulting traffic matrix over the entire period. As one can see from the figure, in this web application, Wordpress nodes do not communicate with each other. In addition, since the client requests are load-balanced in a round-robin fashion, the overall traffic is evenly distributed over the Wordpress nodes. On average, the total size of flows for 5-min intervals is 8.2 GBytes (min: 6.1GB, max: 8.5GB).

### 3.4.1.2 Map-Reduce Tenant

We deployed a Hadoop cluster consisting of 1 master node and 10 slave nodes for map-reduce-style tenant traffic generation. For realistic workload generation, we batch processed 100 jobs with the distribution of per-job input size close to that of the CloudEra customer (CC-e) and Facebook (FB-2010) introduced by Chen et al. [23], which follows a lognormal distribution with logarithmic mean $16.166$ (≃10MB). Surprisingly, this generated a traffic pattern very similar to that of a Microsoft Datacenter [49], which is mainly used for Map-Reduce jobs. Figure 3.6 shows the resulting distribution of duration of flows in one of the slave nodes.

It took around 80 minutes to finish the all jobs, so we could collect total 16 traffic matrices at 5-minute intervals. Figure 3.5(b) visualizes the traffic matrix of the map-reduce-style tenant we generated. Different from the web service tenant, all map-reduce nodes exchange traffic with each other and the sizes of exchanged traffic are erratic. On average, the total size of traffic per 5-minute interval is 3.1 GBytes but the variance was large (min: 0.1GB, max: 14.9GB).

![Figure 3.5](image-url): Traffic matrices of tenants. (a) Each GW, DB, LB and W stands for Gateway, Database, Load Balancer, and Wordpress node. (b) Each M and S means Master and Slave.
Figure 3.6: Cumulative distribution of flows. The flows are observed at the Slave 0 node while running the Map-Reduce workload emulating the CloudEra and Facebook jobs [23]. Other slave nodes showed the identical pattern. More than 70% of the flows last less than ten seconds. Also, 50% of the bytes are in flows lasting less than 1.3s, which could be shifted to 25s similar to the Microsoft Datacenter [49] when we changed the lognormal mean of the job size to 1GBytes.

3.4.1.3 Synthesis into Datacenter

Now we can simulate any virtual datacenter with multiple tenants, each of which may have each traffic matrix we measured. For the evaluation of general performance of Polygravity (Section 3.4.2) and its noise cancellation (Section 3.4.3), we created a virtual datacenter consisting of 60 virtual machines, 20 hosts, 4 edge switches, 2 aggregation switches, 1 core switch, and 1 gateway to the Internet. For the scalability evaluation (Section 3.4.4), we gradually increase the scale of the datacenters.

A traffic pattern of a datacenter can be also influenced by its virtual machine placement policy. To answer the question ‘Does virtual machine placement policy influence the performance of estimation algorithms?’, we tested two contrasting placement policies: affinity policy — ‘placing a VM topologically close to the other VMs belonging to the same tenant’ — and anti-affinity policy — ‘placing a VM in a host that does not host any VMs belonging to the same tenant’. The affinity policy is preferable for improving the network throughput among the VMs, and the anti-affinity policy is better for failure resistance.

For web-service-style tenants, we also investigated the influence of tenant-side configuration of

---

5For the aggregated switches for achieving large bandwidth (e.g., Fat-tree structure with ECMP), our model regards an aggregated group of switches as one switch. Likewise, any aggregated group of links is regarded as one link.
security groups (or virtual topology). As explained in Section 3.4.1.1, the web service uses a three-tier architecture where each node communicates only with specific peers. In a cloud environment, a cloud tenant can setup his virtual topology and/or security groups to reflect such a communication pattern (e.g., for dividing broadcast domains or enhancing security). We tested three possible security group setups – flat, tiered, and point-to-point as illustrated in Figure 3.7. The flat security group setup simply allows every possible communication among the VMs and the external gateway, which makes the system vulnerable to attack since every single node is reachable from outside as well as each other. The tiered setup distinguishes VMs by their type, groups them together, and limits communication by defining the accessible groups for each group. The point-to-point setup strictly limits unnecessary communication, and is the most robust against propagation of attacks. Since the cloud provider can access metadata describing tenants’ virtual topology and security group setup, the cloud provider can readily reflect this information in the condition function \( h(l_x, l_y) \) of Polygravity.

In the map-reduce environment, since all nodes may communicate with each other according to the job assignment by the master node, it is not easy for cloud providers to capture application-specific traffic patterns without special introspection or support from the tenants. As an alternative, we applied preestimation models introduced in Section 3.3.2.3 exclusively for map-reduce-style tenants using host-based, n-out-of-N packet sampling with various sampling rates.

![Figure 3.7: Different levels of access control setups for web service. Each green, yellow, blue and red node refers to Gateway, Load Balancer, Wordpress node, and Database, respectively.](image-url)
3.4.2 Performance

To evaluate the performance of Polygravity, we conducted evaluations with various algorithms against various combinations of tenant styles and security group setups. The evaluated algorithms include tomogravity [104] and ATME-PB [44] as well as Polygravity. Tomogravity is identical to the combination of simple gravity and LSM/IPF. ATME-PB refines the gravity model by excluding traffic across tenants and applies the Nonnegative Least Square (NNLS) method after scaling down the initial gravity result as much as 20%. Thus, the initial model of ATME-PB is identical to the customized gravity model of Polygravity with flat security group setup when we can ignore the internal sink/source traffic. For a fair comparison, we additionally tested the NNLS in the way ATME-PB suggests. For each combination of tenant styles, security group setups, and placement policies, we generated 10 different data sets.

As metrics to compare the performances of different algorithms, we reused the Root Mean Square Error (RMSE) and the Root Mean Square Relative Error (RMSRE) used by Zhang et al. [104]. RMSE and RMSRE are defined as below:

\[
RMSE = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (\hat{x}_i - x_i)^2}
\]

\[
RMSRE(T) = \sqrt{\frac{1}{n_T} \sum_{i \in \{x_i > T\}}^n \left(\frac{\hat{x}_i - x_i}{x_i}\right)^2}
\]

where \(\hat{x}_i\) is the estimation of flow \(i\), \(x_i\) is the ground truth, and \(n_T\) is the number of flows greater than the threshold \(T\). We adjusted \(T\) to cover the largest flows comprising 75% of the entire traffic as Zhang et al. [104] did.

3.4.2.1 General Performance

In Table 3.1(a), we present the resulting traffic matrix errors against datacenters running 5 web-service-style tenants with different placement policies. As we can see from the result, Polygravity starts to show significant performance improvement once we applied the domain knowledge about the “three-tiered security group setup.” What then resulted in the tipping point from the “flat” to the “tiered”? For an intuitive explanation consider Figure 3.8, which visualizes traffic matrices. If we compare the initial estimation results of both (Figure 3.8(b) and (d)), one can see that both of the domain-knowledge runs yield similar results except for the lines along the gateway nodes (the top line and the leftmost line) and the lines from load balancers to the Wordpress nodes (the second bottom lines in each tenant square) in (b). In other words, the condition function of the ‘tiered’ case prevents the gravity model from distributing traffic to unlikely existing flows along the lines,
Table 3.1: Performance of various algorithms. RMSREs are computed on the largest 75% of the entire traffic. RMSEs are in Mbps. The fractions next to algorithm names refer sampling rates.

(a) A datacenter running 5 web-service-style tenants

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Traffic Matrix Errors</th>
<th>Antiaffinity</th>
<th>Antiaffinity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RMSE</td>
<td>RMSRE</td>
<td>RMSE</td>
</tr>
<tr>
<td>Tomogravity</td>
<td>682.93</td>
<td>70.52%</td>
<td>703.83</td>
</tr>
<tr>
<td>ATME-PB</td>
<td>568.91</td>
<td>44.26%</td>
<td>532.98</td>
</tr>
<tr>
<td>Polygravity-Flat</td>
<td>485.93</td>
<td>36.06%</td>
<td>480.77</td>
</tr>
<tr>
<td>Polygravity-Tiered</td>
<td>0.29</td>
<td>0.01%</td>
<td>1.66</td>
</tr>
<tr>
<td>Polygravity-Point-to-Point</td>
<td>0.24</td>
<td>0.00%</td>
<td>1.24</td>
</tr>
</tbody>
</table>

(b) A datacenter running 5 map-reduce-style tenants

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Traffic Matrix Errors</th>
<th>Antiaffinity</th>
<th>Antiaffinity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RMSE</td>
<td>RMSRE</td>
<td>RMSE</td>
</tr>
<tr>
<td>Tomogravity</td>
<td>143.10</td>
<td>58.64%</td>
<td>181.23</td>
</tr>
<tr>
<td>ATME-PB</td>
<td>139.33</td>
<td>55.94%</td>
<td>144.08</td>
</tr>
<tr>
<td>Polygravity</td>
<td>119.09</td>
<td>49.21%</td>
<td>127.46</td>
</tr>
<tr>
<td>Sampling-(\frac{1}{1000})</td>
<td>89.81</td>
<td>48.39%</td>
<td>93.72</td>
</tr>
<tr>
<td>Polygravity-(\frac{1}{1000})</td>
<td>58.76</td>
<td>30.14%</td>
<td>64.68</td>
</tr>
<tr>
<td>Sampling-(\frac{1}{300})</td>
<td>43.35</td>
<td>21.14%</td>
<td>44.09</td>
</tr>
<tr>
<td>Polygravity-(\frac{1}{300})</td>
<td>36.49</td>
<td>15.93%</td>
<td>37.48</td>
</tr>
<tr>
<td>Sampling-(\frac{1}{100})</td>
<td>24.36</td>
<td>13.27%</td>
<td>25.44</td>
</tr>
<tr>
<td>Polygravity-(\frac{1}{100})</td>
<td>19.56</td>
<td>9.90%</td>
<td>21.35</td>
</tr>
</tbody>
</table>

Figure 3.8: Sample traffic matrices in a datacenter with 5 web service tenants and the affinity placement policy. (b) and (d) are the initial traffic matrices for (c) and (e).
but that of “flat” does not. This may make a big difference in terms of the position in the least square’s constraint subspace (i.e., their closest solutions on the constraint subspace are far from each other) and the difference can be signified as they go through the refinement. A similar pattern of performance improvement was observed in the study of Zhang et al. [104]: they could almost double the accuracy of tomogravity by applying domain knowledge about access links and peer links (and, presumably, by excluding traffic among peer links). We argue that the key for performance improvement of traffic estimation is finding such tipping-point domain knowledge.

Finding such domain knowledge is not always obvious, as in the case for map-reduce-style tenants. In a map-reduce cluster, every node communicates with every other node and the traffic among them is not necessarily evenly distributed as shown in Figure 3.9(a). Since the domain knowledge we used for the map-reduce-style tenant is “flat,” meaning “every node can communicate each other and the traffic is distributed proportional to each of their input and output traffic,” Polygravity hardly captures the erratic patterns of the distribution of the traffic as Figure 3.9(c). However, when supplementary model for traffic matrix estimation (such as sampling) is available, Polygravity could effectively integrate it into the model and improve the estimation result as shown in Figure 3.9(d) and (e). Especially for n-out-of-N sampling, we observed 25–35% reduction of sampling-based estimation’s relative error through Polygravity. Table 3.1(b) summarizes the results.

As we can see from both Table 3.1(a) and (b), the placement policy did not show significant influence on the performance of Polygravity, though the Affinity policy showed slightly better performance in general.

3.4.2.2 Heterogeneous Tenants

In a real-world cloud environment, the types of tenants are unlikely to be the same over the datacenter. Therefore, we can naturally ask if Polygravity can still perform well if the target

![Figure 3.9: Sample traffic matrices in a datacenter with 5 map-reduce tenants and the affinity placement policy. (b) is the initial traffic matrices for (c), and (d) is the preestimation model for (e).](image)
datacenter hosts heterogeneous tenants with different granularity of domain knowledge. To answer to this question, we conducted another experiment with a datacenter hosting 2 map-reduce-style tenants and 3 web-service-style tenants with two different placement policies. (For the web-service-style tenants, we used ‘tiered’ security group setup.) Table 3.2 shows the result.

First of all, we notice that Polygravity still showed good estimation performance for the tenants who offer more fine-grained domain knowledge. This property of Polygravity has a significant implication, especially for practical utilization of Polygravity in multitenant environment – the quality of estimation can be isolated for each tenant. Also, if a cloud user wants better quality of estimation service, he can help the cloud provider by offering more fine-grained domain knowledge, such as carefully set security groups or application-level metadata.

Another noticeable feature is that the case with the affinity policy generally showed better performance for the tenants with fine-grained domain knowledge. We can understand the difference to be the result of the LSM. If the flows from different tenants are less entangled at the link level, the least square computation does not need to adjust the flows of a tenant in a large due to the flows of another tenant, so it can minimize the noise from the tenants with coarse-grained domain knowledge. This can be verified from the initial gravity results of the two placement policies, where the difference in the error was negligible.

Table 3.2: Performance of Polygravity with heterogeneous types of tenants. Each MR and WS stands for a map-reduce-style tenant and a web-service-style tenant. RMSEs are in Mbps. RMSREs are computed on the largest 75% of the traffic. For the per-tenant performance, RMSEs and RMSREs are computed based on the traffic of each tenant.

<table>
<thead>
<tr>
<th>Tenant</th>
<th>Traffic Matrix Error</th>
<th>Affinity</th>
<th>Antiaffinity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RMSE</td>
<td>RMSRE</td>
<td>RMSE</td>
</tr>
<tr>
<td>Total</td>
<td>113.59</td>
<td>21.19%</td>
<td>127.80</td>
</tr>
<tr>
<td>MR1</td>
<td>537.36</td>
<td>46.13%</td>
<td>532.92</td>
</tr>
<tr>
<td>MR2</td>
<td>374.82</td>
<td>58.51%</td>
<td>508.50</td>
</tr>
<tr>
<td>WS1</td>
<td>3.57</td>
<td>0.16%</td>
<td>7.44</td>
</tr>
<tr>
<td>WS2</td>
<td>3.81</td>
<td>0.15%</td>
<td>7.75</td>
</tr>
<tr>
<td>WS3</td>
<td>3.92</td>
<td>0.08%</td>
<td>23.37</td>
</tr>
</tbody>
</table>
To evaluate the performance of the inner gravity estimation (IGE) and the broadcast noise cancellation (N/C) of Polygravity, we injected various tenant-level multicast traffic into a datacenter with 5 web service tenants and compared the performance of Polygravity with and without IGE and noise cancellation. Specifically, we varied three different factors of noise traffic: the total size of interior source traffic (SourceSize), the number of destination VMs for each multicast (NumDsts), and the number of multicast-generating tenants (NumTenants). While varying each factor, we fixed other factors at their default values (SourceSize=32GB, NumDsts=10, NumTenants=5). Each experiment was repeated 10 times with different data sets, both in a datacenter with the affinity policy and in another with the antiaffinity policy.

### 3.4.3.1 SourceSize

Figures 3.10 (a1) and (b1) show the changes of relative errors as the size of each multicast flow exponentially increases, from 728KB (introducing 32MB of traffic from interior sources, and additional link-level traffic as much as 0.03% of the nonnoise traffic) to 1GB (introducing 32GB of traffic from interior sources, and additional link-level traffic as much as 25% of the nonnoise traffic). As one can see from the figures, IGE and N/C consistently suppress the error due to broadcast. Especially when the size of broadcast traffic was large (greater than 1% of the nonbroadcast traffic) in the datacenter with the affinity policy, IGE and N/C consistently decreased the relative errors of nonnoise flows by $\frac{2}{3}$ (the gap between two lines in Figure 3.10(a1)). Interestingly, in the datacenter with the antiaffinity policy, the vanilla Polygravity (i.e., Polygravity without IGE and N/C) showed better noise tolerance, so IGE and N/C could help to decrease the RMSRE only by $\frac{1}{2}$ to $\frac{1}{3}$.

### 3.4.3.2 NumDsts

Even if the total size of the interior sources is the same, the distribution of the interior sources can vary depending on the form of multicast. For example, although both a 2MB multicast flow destined to two nodes and a 1MB multicast flow destined to three nodes will introduce 2MB of traffic from interior sources, the first one actually generates 2MB of traffic from a single interior source and the second one introduces 1MB of traffic from each of two interior sources. We changed the distribution of interior sources by changing the number of destinations of each multicast flow while keeping the total size of interior source traffic to be 32GB, and observed the performance of IGE and N/C. Figure 3.10 (a2) and (b2) show the results. In a nutshell, the more widespread the interior
sources are, the more noise-tolerant Polygravity was, but the performance improvement by IGE and N/C was consistent regardless of the distribution: 30%–60% smaller RMSRE with the affinity policy and 10%–20% smaller with the antiaffinity policy. Similar to the case of varying SourceSize, the vanilla Polygravity showed better noise tolerance with the antiaffinity policy, which decreased the contribution of IGE and N/C.

### 3.4.3.3 NumTenants

Another way to change the distribution of the interior sources is changing the number of multicast flows. This time, we changed the number of multicast flows by limiting the number of tenants generating the multicast flows. (The number of destinations and the total size of interior source traffic are fixed.) For the affinity case, the result was similar to the case of varying NumDsts. However, in the antiaffinity case, vanilla Polygravity showed stronger noise-tolerance, even outperforming the Polygravity with IGE and N/C for some cases (NumTenants=1 and 2).
In summary, we observed that IGE and N/C consistently improved the performance of Polygravity when there exists widely distributed broadcast noise. This can be explained by the nature of the gravity models in Polygravity, which assumes the interior sinks/sources are proportionally distributed over the network interfaces and the sinks/sources are proportionally mapped with other network ends. In addition, the vanilla Polygravity showed stronger noise-tolerance under the anti-affinity policy, though there was still some room for improvement by IGE and N/C. Presumably, this is because the noise traffic could be more evenly scattered across all flows in the affinity case, which decreases the root mean square value.

### 3.4.4 Scalability

To evaluate the scalability of Polygravity, we gradually scaled up the target datacenter as shown in Table 3.3 and measured the computation time and accuracy for both Polygravity and tomogravity. Figure 3.11 shows the changes of computation time as the target datacenter scales up. The computation time of LSM was almost identical for both (Polygravity had a slightly larger time due to the augmented gravity domain.) For the initial gravity modeling, Polygravity’s case takes obviously longer than tomogravity’s simple gravity model. The most dramatic difference between the two algorithms was shown in the IPF step. This is because tomogravity’s estimation result was generally too far from the ground truth, which leads to an excessive number of iterations at the IPF step. As one can see from the figure, the main scalability bottleneck for Polygravity is LSM (Singular Value Decomposition, to be specific), which is essentially the same for tomogravity in ISP network. One can reduce the actual computation time of SVD through two approaches: first, by making use

<table>
<thead>
<tr>
<th>Scale</th>
<th>Flows</th>
<th>Links</th>
<th>VMs</th>
<th>Hosts</th>
<th>EdgeSW</th>
<th>AggSW</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>930</td>
<td>44</td>
<td>30</td>
<td>10</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3660</td>
<td>87</td>
<td>60</td>
<td>20</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>8190</td>
<td>130</td>
<td>90</td>
<td>30</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>14520</td>
<td>173</td>
<td>120</td>
<td>40</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>22650</td>
<td>216</td>
<td>150</td>
<td>50</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>32580</td>
<td>259</td>
<td>180</td>
<td>60</td>
<td>12</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>44310</td>
<td>302</td>
<td>210</td>
<td>70</td>
<td>14</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>57840</td>
<td>345</td>
<td>240</td>
<td>80</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>73170</td>
<td>388</td>
<td>270</td>
<td>90</td>
<td>18</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>90300</td>
<td>431</td>
<td>300</td>
<td>100</td>
<td>20</td>
<td>10</td>
</tr>
</tbody>
</table>
of distributed SVD algorithms with more computation resources, and second, by reducing the size of the matrix \( m \times n \). As future work, we are exploring an effective way to reduce \( n \) by grouping uninteresting flows without sacrificing the accuracy for other flows.

Regarding the accuracy for the algorithms, regardless of scale, both of the algorithms showed highly consistent results: 77% to 82% of RMSRE for tomogravity, and 0.12% to 0.14% for Polygravity.

### 3.4.5 Measurement Error Tolerance

Measurement error in SNMP counters is very common, so one might have a practical concern regarding the impact of this measurement error on the performance of Polygravity. Zhang et al. [104] showed that tomogravity has robust error tolerance against this type of measurement error by inducing errors to the tomogravity model. To see if Polygravity inherits this feature from tomogravity, we conducted the same experiment.

To be specific, we first generated an error term \( \epsilon \):

\[
\epsilon = x \ast N(0, \sigma)
\]  

where \( \ast \) is an element-wise multiplication operator and \( N(0, \sigma) \) is a vector with random entries following a normal distribution with mean 0 and standard deviation \( \sigma \). We then induced the error \( \epsilon \) to the link vector:

\[
x_{err} = x + \epsilon
\]
We ensured the non-negativity of $x_{err}$ by changing negative values in $x_{err}$ to 0. Table 3.4 shows the performance of Polygravity under different levels of noise. Here, we used a datacenter hosting five web-service-style tenants with the affinity placement policy, and each experiment was repeated over 10 different data sets. The result was identical to that of the tomogravity: the measurement-level errors proportionally degrade the accuracy of Polygravity.

3.5 Discussion

Polygravity is designed for continuous traffic matrix monitoring in a multitenant datacenter. In this section, we discuss some practical concerns for deploying Polygravity.

3.5.1 Sampling and Other Estimation Techniques

Polygravity is not meant to replace all benefits of sampling. For instance, as we saw in Figure 3.9(d), sampling alone shows great performance for the elephant flow detection. However, to increase its coverage, we must increase the sampling rate by some orders of magnitude, as we saw in Table 3.1(b). An advantage of Polygravity over the sampling-only approach is that we can selectively apply sampling techniques. For example, when a cloud administrator wants to estimate the entire traffic matrix of the datacenter with smaller relative error, the administrator can apply a high sampling rate to just the nodes with coarse-grained domain knowledge.

This advantage of Polygravity is applicable to other estimation and/or measurement techniques. For instance, when a cloud administrator wants a precise traffic matrix but domain knowledge for certain flows is imprecise, he may set the SDN rules for the flows, collect precise flow counts for them, and simply reflect the measurement result as a preestimation model of Polygravity. This feature of Polygravity allows cloud administrators to flexibly deploy any hybrid solution.

Table 3.4: Measurement error tolerance of Polygravity. RMSEs are in Mbps. RMSREs are computed on the largest 75% of the traffic.

<table>
<thead>
<tr>
<th>Noise Level</th>
<th>traffic matrix errors</th>
<th>link errors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RMSE</td>
<td>RMSRE</td>
</tr>
<tr>
<td>noise free</td>
<td>0.29</td>
<td>0.01%</td>
</tr>
<tr>
<td>$\sigma = 0.01$</td>
<td>18.45</td>
<td>1.72%</td>
</tr>
<tr>
<td>$\sigma = 0.02$</td>
<td>29.98</td>
<td>3.05%</td>
</tr>
<tr>
<td>$\sigma = 0.04$</td>
<td>49.65</td>
<td>5.46%</td>
</tr>
</tbody>
</table>
### 3.5.2 Interior Sinks/Sources

Considering Section 3.4.3, one might wonder if there exists such a high volume of traffic from interior sinks and sources in a datacenter. As a practical example, in the CloudLab datacenters [75], the virtual machine and bare metal images are transferred to the distributed host machines through Frisbee [42], which internally uses switch-level multicast to reduce the amount of traffic in the datacenter. Through our experience of traffic analysis on one of the CloudLab clusters, we could observe large traffic from interior sources due to Frisbee, especially when there were many VMs starting off. Tenant-level multicast is another source of such interior source traffic. Unless a VM itself changes multicast to unicast, a multicast flow unavoidably induces a branching of the flow to multiple destinations, either at switch devices or at host machines (e.g., OpenStack Neutron ML2 [74]).

Regarding the interior sinks, the most practical example is packet loss due to a cloud’s security group (distributed firewalls) [4, 67]. When we estimate VM-to-VM traffic, host machines work as intermediary network devices. However, if a flow from one VM to another is dropped by a security group rule, it does not go through the virtual interface between the destination VM and its host machine. In this case, the flow cannot be mapped to the VM-to-VM flow, but to a flow between the source VM and the destination VM’s host machine. This makes the host machine seem to be both an intermediary device and an end host, which is an interior sink in our model. Note that this type of noise flow does not need to be noise-canceled, since the number of such flows is very small in comparison to the number of possibly existing multicast flows.

### 3.5.3 Usability

As mentioned earlier, an accurate end-to-end traffic matrix can provide a great deal of help in many different network problems: traffic engineering [2], virtual machine scheduling [57], network design [73], capacity planning, and failure detection [38]. However, traffic matrix estimation may not be suitable for some other types of problems. For instance, Polygravity can be overkill for cloud billing, because we do not need to know the size of every end-to-end flow but rather the aggregated amount of traffic (e.g., aggregated amount of traffic in-out/to-from a datacenter for each tenant in AWS [3]). Likewise, Polygravity may be insufficient for problems that require a finer-grained level of traffic information (e.g., protocol, size and interarrival time of packets for datacenter security analysis [21, 101]). In addition, if the target problem is sensitive to the accuracy of the estimation result, the administrator needs to carefully compare the expected accuracy of the Polygravity model.
in the target network and the required accuracy for the problem. Since precise comparison of these two accuracies is still an open problem, as future work, we plan to apply Polygravity to some of the aforementioned network problems to answer the question as well as to see the impact of our solution.

3.6 Conclusion

In this chapter, we presented Polygravity, a traffic matrix estimation algorithm for a multitenant datacenter via coarse-grained, link-level measurement data and using different types of domain knowledge. Through our evaluation, we showed Polygravity can estimate the traffic matrix with less than 1% average relative error with fine-grained domain knowledge. In addition, when provided domain knowledge is coarse-grained, Polygravity’s estimation has $\frac{1}{3}$ smaller relative error with the assistance of sampling than a sampling-only approach. Polygravity is especially suitable for a multitenant environment since it can show relatively clean performance isolation for each tenant with heterogeneous domain knowledge. Our scalability evaluation showed that Polygravity consistently performs well regardless the scale of target datacenter.
CHAPTER 4

TENANT-SIDE VISIBILITY THROUGH A SIDE CHANNEL

In this chapter, we revisit tenant-side visibility problem and explore how unintended visibility into the cloud platform can be obtained and exploited.

4.1 Introduction

Resource sharing is a fundamental part of cloud computing. By multiplexing virtual resources (e.g., virtual machines, virtual networks, virtual firewalls, etc.) across an infrastructure, a cloud provider maximizes resource utilization of the infrastructure and offers cloud users flexible scaling of virtual environments with minimal costs.

However, shared resources also cause interference among cloud tenants and can even be exploited as information leakage channels by malicious users to make critical security breaches. For example, if an attacker’s virtual machine (VM) can be successfully placed in a physical machine hosting victim VMs, the attacker VM can exploit such information leakage channels to detect if it is co-resident with a victim [77, 89, 99], to degrade the performance of a victim [88], or to break the virtual isolation and steal confidential information from compromised and noncompromised victims [91, 97, 102, 95, 103, 100, 53]. These side/covert-channel attacks are still being actively studied and becoming more feasible and practical.

The underlying mechanisms exploited by the previously studied information leakage channels were mostly limited to hardware architecture-level mechanisms managing a specific set of hardware resources: CPU, L2/L3 caches, memory, and network devices. However, under the hood of a cloud platform, cloud tenants share resources not only at the hardware level but also at the software level such as processes, threads, kernel modules, and networks of cloud management systems. Especially in cloud management planes, it is commonplace for a single software instance to process multiple requests from different tenants (both at the central cloud controllers and at the distributed cloud management components). For instance, for two co-residing VMs, if each of their users makes a
request to connect each instantiated VM connected to a virtual network, the two requests will go through the same virtual network management software instances such as a local cloud management service, a local OpenVSwitch service, and a netfilter kernel module, as illustrated in Figure 4.1. More importantly, these requests may share some parts of their execution paths as illustrated in steps 5, 6, and 7 of Figure 4.1 (e.g., due to batch-processing mechanisms for performance optimization). Therefore, the processing times of the two requests may mutually influence each other.

Our key insight is that if a VM can keep track of the processing times of its own infrastructure-level requests, it can obtain footprints of co-resident VMs’ infrastructure-level information — e.g., virtual firewall update times or start/end times of co-resident VMs. Since this type of information is not obtainable by previously known side channels, the potential impact of this new type of side channel can be significant.

It is challenging to monitor the infrastructure-level activities from a cloud user’s side because a user does not have visibility into the cloud infrastructure-level events. One may wonder if the user can utilize the data provided by cloud providers such as APIs for the current virtual resources or event logs [7]. Unfortunately, most of resource-state information offered by the cloud providers is

![Figure 4.1: Resource sharing of two requests. Two requests for two different VMs go through the same processes. Though they come in separately to the RPC front-end, they are merged at the core logic thread of local virtual network management service and processed together for the remaining steps.](image-url)
not based on the actual event times at the edge, so the states of virtual resources provided by the cloud provider may not be consistent with their actual states at the edge [16].

In this chapter, we introduce a novel technique to exploit cloud management mechanisms as an information leakage channel. We manipulate the timing of infrastructure events by requesting, via the cloud provider’s API, specific types of modifications to a tenant’s virtual firewall rules (often called ‘Security Groups’). We detect these effects using specially crafted probe packets that monitor changes to firewall state. To demonstrate the utility of this approach, we implement two different classes of firewall-based covert channels for communication between otherwise-isolated tenants and show a new class of a side channel exploiting the firewall to eavesdrop infrastructure level events in OpenStack.

To summarize, our contributions include the following:

- We illustrate that the software architecture processing requests from different tenants is exploitable as an information leakage channel through a measurement study in OpenStack’s network management stack. To the best of our knowledge, this is the first work that shows the exploitability of shared software resources in cloud management planes as an information leakage channel (Section 4.3).

- We devise a novel infrastructure-level firewall monitoring mechanism to exploit virtual firewalls as an information leakage channel (Section 4.4).

- We demonstrate the exploitability of the virtual firewall by implementing two different firewall-based covert channels between two isolated VMs (Section 4.5). We also demonstrate a proof-of-concept of a firewall-based side channel to eavesdrop the infrastructure-level events (Section 4.6).

- We discuss strategies for mitigating these channels (Section 4.7).

### 4.2 Background

In this section, we briefly introduce the internal architecture of OpenStack to show the fundamental mechanism of the firewall-based information leakage channel.

In OpenStack, a VM has a virtual firewall, which consists of a list of rules specified in the security groups that the VM belongs to. The ‘security group rules’ of a security group are instantiated into firewall rules when the security group is applied to a specific VM. OpenStack implements the virtual
firewall using *Linux Iptables* in each host machine. Specifically, in each host machine, there is an OpenStack component called Neutron-OpenVSwitch-agent that manages most of the host-side network resources, including the iptables. When a user requests the application of a security group to a VM, the cloud controller sends this request to the machine hosting the VM, and the agent in the host retrieves the request and correspondingly updates the iptables.

The main part of the Neutron-OpenVSwitch-agent is an infinite loop that processes RPC requests. The agent does not immediately process a request when the request is received. Instead, it collects requests and periodically iterates over the RPC loop to process the collected requests at once, which is a general strategy for service-oriented architectures to improve throughput and end-to-end latency [26]. The loop iteration period can be configured by changing the value of the variable `polling_interval` in a file called `openvswitch_agent.ini`, for which the default value is 2 seconds. (Let $n$ be the current `polling_interval` value for convenience.) However, this variable does not guarantee that the RPC loop takes exactly $n$ seconds: instead, the system guarantees that each iteration takes at least $n$ seconds. This is internally implemented by making the process sleep at the end of each iteration if the iteration took shorter than $n$ seconds, as shown in the following code:

```python
while True:
    start = now()
    # process the enqueued tasks
    elapsed = now() - start
    if elapsed < polling_interval:
        sleep(polling_interval - elapsed)
```

Therefore, when a user makes a request to update a VM’s firewall, the actual update of the corresponding iptables is unlikely to happen immediately. If the request arrives just before the next iteration starts, the corresponding iptables rule will be created very soon, but it may take a very long time if the previous iteration is not finished. Naturally, if two or more users make requests within the same period and their VMs are hosted by the same physical machine, the requests will be processed together as illustrated in Figure 4.1, and the latency of the requests will be influenced by each other. Under this situation, if a VM can know the execution duration of the current iteration (i.e., `elapsed`
in the code above\(^1\), it may also know the impact from the request made by other VM. Of course, in reality, a VM cannot directly know the execution durations of the process running in the host. However, as explained earlier, since updating the VM’s firewall also happens at some point of the iteration, if the VM can know an interval between two consecutive firewall-updating events, the VM may utilize this as an alternative to the execution duration of the process. We call this interval from an iptables update to the very next iptables update an “Epoch.” As illustrated in Figure 4.2, an Epoch may or may not be close to the actual execution duration.

In Section 4.3, we first assess the magnitude of impact of user requests on the execution duration of the shared RPC loop and show the feasibility of exploiting Epoch as an information leakage channel.

### 4.3 Measurement Study

Since every request received by the Neutron-OpenVSwitch-agent is processed at some point of the RPC loop, it is obvious that every single request has some impact on the execution duration of an iteration of the RPC loop. However, the impacts of requests may appear in various patterns of Epochs as shown in Figure 4.3. For example, in the case of the second iteration in the figure, the request increased the elapsed time after updating the iptables, but was counter-balanced by the sleep time, so the impact could not be observed between Epochs. In contrast, in the case of the seventh iteration, the request increased the elapsed time before updating the iptables, and it made Epochs oscillate noticeably. In addition, as one can see from the next three requests (the 12th, 17th, and 22nd iterations), the impacts of requests on Epochs may appear differently from the impacts on the total

---

\(^1\)To avoid confusion, in the rest of this chapter, we use the term elapsed time only to refer the value of the variable elapsed.
Fig. 4.3: Various impacts of requests on Epochs. Before and After are relative to when the iptables are updated. Total is the sum of Elapsed and Sleep.

execution duration of the iterations. The impact of a request on the elapsed time before updating the iptables can influence the size of the current Epoch, but the impact on the elapsed time after updating may influence on the size of the next Epoch only if the total execution time of the current iteration exceeds the polling interval.

In this section, through measurement taken in a cloud running OpenStack Mitaka, we show the impact of requests and their combinations on the elapsed times before and after updating the iptables. To measure the actual execution duration of the RPC loop, we modified the source code of Neutron-OpenVSwitch-agent and Iptables to print timestamps at several points in the RPC loop.\(^2\) We first analyze the general impact of different user-level requests on the elapsed times in Section 4.3.1. In Section 4.3.2, we introduce some specific ways to permanently increase the elapsed times. The type of all nodes used for this measurement study was Emulab \([93]\) d710 \([87]\).

### 4.3.1 One-Time Impact

We first measured the change in the elapsed times as we made different requests to the shared process. Especially, we measured the elapsed times for four different firewall-related requests: adding a rule, deleting a rule, attaching a security group, and detaching a security group. For the case of

\(^2\)This modification helped our measurement study, but is not necessary to exploit the side channel “in the wild.”
adding and deleting a rule, we first made a security group for a VM and measured the time taken to add a rule to (or delete a rule from) the group. Note that, since the Epoch can be properly monitored by a VM only if the VM makes some changes in its own firewall at every iteration, understanding elapsed times of these firewall-related requests is important to build firewall-based information leakage channels. For the case of attaching and detaching a group, we made a group with one rule and measured the time for making the VM attached to (or detached from) the group. One of the interesting features of these requests is that the adding-a-rule request and the attaching-a-group request yield the same result in the iptables: i.e., for both cases, the iptables chain for the VM will gain a new rule. Likewise, the deleting-a-rule request and the detaching-a-group request also have the same result.

The first four bars in Figure 4.4 show the elapsed times for the four different requests made by the receiver VM. (The sleep time is not included.) For reference, without any requests, the

![Figure 4.4](image)

**Figure 4.4**: Average and standard deviation of elapsed time for requests and their combinations. Each `add`, `del`, `atch`, and `dtch` refers to `add`, `delete`, `attach` and `detach` request. Each experiment was repeated 100 times.
elapsed time was only around 130 milliseconds. As one can see from the figure, each iteration took around 1,200 milliseconds for adding/deleting a rule, but the elapsed time increased by 1,200 milliseconds for attaching/detaching a group (400 milliseconds for before and 800 milliseconds for after). We also measured the change in elapsed time when multiple requests are processed together within the same iteration. Making two addition/deletion requests during the same Epoch was not noticeably different from making a single request, as shown in the next three bars in Figure 4.4. Likewise, when we combined one addition/deletion request with one of attachment/detachment request, the elapsed time was similar to a single attachment/detachment request. However, when two security group attachment/detachment requests were made in the same Epoch, the elapsed time was increased by 1,100 milliseconds (250 milliseconds before and 850 milliseconds after).

From this result, we can see that in order to monitor other tenants' activity (reading from the side channel), addition/deletion requests are more suitable, since their effects are not cumulative with other activity. Likewise, in order to manipulate the elapsed time during Epochs (writing to the side channel), attachment/detachment requests are more useful, since their impact is greater and the effects of each individual request are still visible even when combined with other requests.

However, the result also shows potential reliability problems when using attach/detach requests to manipulate Epoch lengths, because they showed high standard deviations when combined with add/delete requests (Figure 4.4 add/atch, add/dtch, del/atch and del/dtch). We analyzed this problem by examining the functions invoked internally in the agent at each iteration. We found that those requests showed high variance in elapsed times not because the elapsed time of the requests themselves are erratic, but because the agent sometimes postponed a portion of the work to the next iteration. One can also see this effect in Figure 4.5, which shows the actual execution durations of the RPC loop while we were measuring the elapsed times for add/atch and del/atch. Here, at the 78th and 81st iterations, one can see the requests arrived during those iterations (from the fact that the iptables were updated), but the time-consuming tasks were postponed to the next iterations; the execution durations of the 79th and 82nd iterations were increased instead. Though this happens less frequently as we combine more requests, we could not find a way to completely remove this phenomenon, and thus have to account for it in the construction of our side channel.

We also found the elapsed time for requests can be affected by rules that are already in place. For the add/delete requests, as we increased the number of existing rules of the security group to/from which a rule would be added/deleted, the elapsed time for the requests exponentially increased
Figure 4.5: Execution durations of the RPC loop iterations while add+atch and del+dtch requests are arriving. Before and After refers elapsed time before and after the iptables are updated. Before and After only show up only if the iptables are updated in that iteration (i.e., only if either add+atch or del+dtch requests arrived in that iteration). add+atch and del+dtch requests were made repeatedly one after another. The polling_interval was set to be 2 seconds.

(linearly in the before period but exponentially after). Figure 4.6 shows the result. We also changed the number of rules present in the security groups targeted for attach/detach requests. Surprisingly, for the detaching-a-group request, the elapsed time saw little influence from the number of rules (as shown in the right side of Figure 4.7). For group-attach requests, only the elapsed time before updating the iptables was increased as the number of rules in the group increases, and only grows linearly.

From these results, we gained a few insights into exploiting Epochs as an information leakage channel. First, a VM may dynamically manipulate Epochs with various combinations of requests and rules. Second, a VM may extract infrastructure-level information such as types of events (i.e., requests) and the environmental setup regarding the event (e.g., the number of security group rules). In Section 4.5.1, we show how an attacker can make use of different combinations of requests to send and receive different signals through the firewall to build covert channels. Also, in Section 4.6, we show a proof of concept of a firewall-based side channel through which a VM can count the number of VMs in the host.
Figure 4.6: Average and standard deviation (over 100 runs) of elapsed time for `add` and `del` requests as the number of security group rules is increased.

Figure 4.7: Average and standard deviation (over 100 runs) of elapsed time for `atch` and `dtch` requests as the number of security group rules is increased.
4.3.2 Long-Term Impact

While measuring the impact of the number of rules on Epochs, we found another interesting way to increase the size of Epochs: by giving a VM a large number of rules in a specific way, we could increase the elapsed times for requests against other co-residing VMs belonging to a different tenant. We originally observed this phenomenon in an older version OpenStack, Icehouse. In Icehouse, if we increased the number of security group rules for a VM, the elapsed time for any firewall-related requests to the same host were increased. In a newer version of OpenStack, Mitaka, it seemed that this phenomenon had disappeared. Even in Mitaka, however, when we attached a security group with many rules and added one additional rule to the group, it could increase the elapsed time as we saw in Icehouse. Figure 4.8 shows the measurement result for adding a rule in Mitaka. Other requests (deleting, attaching, and detaching) also showed identical results.

According to our source code analysis of OpenStack Mitaka, this phenomenon is because of poorly optimized code; when a new rule is added to an existing security group, the Neutron-OpenVSwitch-agent caches all rules in the group in a list. Later, when there is a request, the cached list is unnecessarily retrieved, and this increases execution duration.

Though this is obviously a bug that should be fixed, it is a very useful feature for attackers, and it illustrates that unless cloud software systems are specifically built to resist observable timing effects,

![Figure 4.8](image)

**Figure 4.8:** Average and standard deviation (over 100 runs) of elapsed time for add request as the number of security group rules used by the long-term impact request increases.
they can be powerful tools for side channels. For instance, if the `polling_interval` is set to be very long, it becomes hard to detect some requests impacting on elapsed time after the `iptables` update (as we saw from the case of the second iteration in Figure 4.3). In this case, if the attacker increases the overall elapsed time to as much as `polling_interval`, there would be no sleep time for any iteration and the signal from those requests becomes clearer. In addition, the attacker may exploit this ‘feature’ to intentionally make processing time for other requests slow. The attacker does not even need to repeatedly make requests to conduct these attacks — all the attacker needs to do is to attach a security group with a large number of rules and add one more rule to it. In Section 4.5.2, we utilize this to implement a broadcast-style covert channel.

**4.4 Monitoring the Virtual Firewall**

In this section, we present our techniques to measure the update times of virtual firewalls. Since the update times inherently tell the execution times of the shared agent, the update times can be used as a side channel for the infrastructure level events in the host.

Figure 4.9 illustrates a basic architecture for monitoring the update times of virtual firewalls, with which the durations of Epochs can be trivially calculated. When a cloud controller receives a request to add a firewall rule allowing a probing packet \( p \), it forwards the request to the local agent, the agent lets the `iptables` add the rule correspondingly, and the rule is finally added to the `iptables` (at time \( t \)). Meanwhile, a series of probing packets \( p \) is being sent to the monitoring VM and, naturally, only the probing packets arriving at the `iptables` after \( t \) can be successfully sent to the VM. Therefore, the VM can estimate the update time of the `iptables` from the arrival times of probe packets. Of course, there are still several questions to be answered for this mechanism to work:

- When and how frequently should the requests and probe packets be sent?

- Who sends the probe packets and who does the rule updating requests?

- What kind of packets and rules can be used for probing?

Since the answers vary depending on the environment and the goal, we discuss different design options in the following subsections. In Section 4.5, we present specific instances of this architecture for two different covert channel scenarios.
4.4.1 UPDATE+PROBE Technique

As we saw in the previous section, if a VM can know the time duration of an Epoch, it can exploit this information to guess the cloud management-level events that happened during the Epoch or to send/receive a signal to co-residing VMs by intentionally making a request to influence the Epoch. A precise measurement on iptables update times is the key for this technique. In the previous section, we could directly measure the update time, since we were operating at the level of the cloud provider. Yet from a cloud user’s perspective (within an unprivileged VM), we can only ‘guess’ the update time of iptables — ‘somewhere after a firewall update request left my node.’

The open-ended probabilistic range of the update time can be narrowed down if the VM can generate a set of packets that exclusively match the updated rule, and observe the exact time when these packets start to be successfully passed through the firewall. For example, if a VM had no rule to allow any ICMP packet in, and if it received an ICMP packet (at $t_2$) after it made a request to add an ingress rule for ICMP packets (at $t_1$), it can know there must have been an iptables update event between $t_1$ and $t_2$. Let a probe packet $p_i$ be a packet that exclusively matches a rule $x_i$. Then, the closer to the actual update time the $p_i$’s firewall passing time is, the tighter the upper bound of the update time estimation the VM can get.

In practical terms, we can obtain such a tight upper bound by continuously sending a series of probe packets ($p_1^1 \ldots p_n^1$) and picking the one that arrives first (say $p_j^1$). Furthermore, we can also probabilistically tighten the lower bound of the update time by estimating the arrival (and drop) time of the previous packet $p_i^{j-1}$. Figure 4.10 summarizes the timeline of this updating and probing step, which we refer to as a UPDATE+PROBE step.

Recall that our goal is not just to know the update time of a firewall rule but to perform continuous
Figure 4.10: Timeline of a UPDATE+PROBE step. \( t(p_j^i) \) refers the arrival time of \( j \)-th probe packet for the rule \( x_i \). \( t_{req}(x_i) \) refers the time when the VM sent the request for the rule \( x_i \). \( t_{up}(x_i) \) refers the update time of the iptables for the requested rule \( x_i \). The prime sign means it is an estimated value.

Algorithm 3 Iterative UPDATE+PROBE method

1: procedure REQUEST(t)
2: while True do
3:     rule ← GetNextRule()
4:     SendRequest(rule) ▷ nonblocking
5:     SendProbes(rule) ▷ nonblocking
6:     sleep(t)
7:     Thread(REQUEST, [polling_interval − \(\varepsilon\)].start())
8: while True do
9:       p ← MonitorFirstProbes()
10:      ReportEvent(p)

monitoring of Epochs. Therefore, we should continuously repeat UPDATE+PROBE steps at least once per iteration of the virtual network management service’s RPC loop. The simplest way to do this is repeating UPDATE+PROBE every \( n \) seconds, where \( n \) is smaller or equal to the minimum polling interval of the RPC loop, as described in Algorithm 3. Here, \textit{GetNextRule()} is a function that returns a new rule that is disjoint from any existing rules; \textit{SendRequest()} is a function that sends a firewall rule update request to the cloud controller; \textit{MonitorFirstProbes()} is a function that monitors arrival time of the target probe packets, returning the first one that arrives (ignoring any subsequent ones, which give us no more timing information); and \( \varepsilon \) is a parameter to adjust the iteration period of REQUEST() (\( \varepsilon \approx polling\_interval − n \) if the time spent for an iteration of REQUEST() is negligible).

Though it is harmless to perform UPDATE+PROBE more than once per RPC loop iteration, doing so will increase the number of API calls against to the cloud controller and may increase the chance to be detected or rate-limited by cloud administrators. Thus, an ideal monitoring mechanism should
Algorithm 4 Reactive UPDATE+PROBE method

1: do
2:   rule ← GetNextRule()
3:   SendRequest(rule) ▸ nonblocking
4:   SendProbes(rule) ▸ nonblocking
5:   p ← MonitorFirstProbe(rule)
6:   ReportEvent(p)
7: while True

Algorithm 5 \( n \)-Reactive UPDATE+PROBE method

1: procedure REQUEST\((t, n)\)
2:   for \( i \in \{1, \ldots, n\} \) do
3:     rule ← GetNextRule()
4:     SendRequest(rule) ▸ nonblocking
5:     SendProbes(rule) ▸ nonblocking
6:     sleep\((t)\)
7:   rule ← GetNextRule()
8:   SendRequest(rule) ▸ nonblocking
9:   SendProbes(rule) ▸ nonblocking
10: while True do
11:   p ← MonitorFirstProbe()
12:   ReportEvent\((p)\)
13:   if IsNewEvent\((p)\) then
14:     Thread\((\text{REQUEST}, \lfloor \text{polling\_interval} / n, n \rfloor).start()\)

perform a UPDATE+PROBE step once per iteration of the RPC loop.

We can design an algorithm that performs UPDATE+PROBE exactly once a RPC loop iteration by letting the next UPDATE+PROBE start as soon as the updating event of the previous request is reported, as described in Algorithm 4. From the algorithm, one can see that the loop is blocked at \( \text{MonitorFirstProbe()} \) so that the method iterates the loop exactly once every time iptables are updated. We name this algorithm \textit{Reactive UPDATE+PROBE method}.

The reactive UPDATE+PROBE method is based on the assumption that the request of the next UPDATE+PROBE step arrives at the RPC front-end before the next iteration of the RPC loop begins. According to the measurement result shown in Section 4.3, the gap between the iptables updating time of the current RPC loop iteration and the start of the next RPC loop iteration was long enough (\( > 800\text{ms} \) with the default polling interval setup) for a round-trip of a update request. However, if the cloud controller is under a heavy load, the round trip time may increase, and the method may miss one (or more) iptables update events and report a large single Epoch (which actually consists of multiple Epochs). As a remedy to this problem, we can modify the reactive UPDATE+PROBE algorithm to perform the UPDATE+PROBE steps \( n \) times, with a gap of \( \text{polling\_interval} / n \) after the
identification of iptables update, as shown in Algorithm 5. Figure 4.11 illustrates a timeline of the method.

To apply the modified reactive UPDATE+PROBE algorithm, it is necessary to know the minimum polling period of the RPC loop. However, the RPC polling period is an infrastructure-level configuration value, so it is not directly known by cloud tenants. For this reason, identifying this polling period needs to be done before the $n$-reactive UPDATE+PROBE algorithm begins. This polling can be measured by using iterative UPDATE+PROBE with a very small polling_period value. If the actual polling period is much larger than the polling_period value we set, multiple UPDATE+PROBE steps will identify the same iptables update event, and the minimum polling period will be as much as the gap between two consecutive iptables update events.

Now that we have defined an algorithm for monitoring epochs, the next practical questions are ‘how can a VM let probe packets go through the firewall and come to itself?’ and ‘how can we keep generating rules that do not overlap with any previous rules?’ To answer to these questions, we need to discuss the details of the target virtual network environment. In the following sections, we review practical virtual network configurations and present a representative system architecture to monitor epochs.

![Figure 4.11: $n$-Reactive UPDATE+PROBE method. $t_{arr}(x_i)$ refers the time when the request for the rule $x_i$ arrived at the edge. The requests for $x_1$ and $x_2$ have arrived by the start of the very next iteration of the RPC loop, but the request for $x_3$ has not. Thus, only $x_1$ and $x_2$ can be probed after the next iptables update. We could get a narrower bound for $t'_{up}(x_1)$ and $t'_{up}(x_2)$ by using the intersection of the two bounds.](image-url)
4.4.2 Deployment

In Section 4.4.1, we have assumed that the VM can send packets through the firewall and observe if they are passed through. The simplest way to do this is letting another VM (say a helper node) send or receive the probe packets, though this can make precise timing difficult. However, a helper node is not always necessary: in many cases, the VM can send probe packets which through the firewall and come back to itself. We call packets with this property ‘boomerang packets’: we define a boomerang packet for a firewall rule $x_i$ as an uniquely identifiable packet that (i) is sent by the source node and (ii) is delivered back to the source node (iii) without bypassing$^3$ the firewall rule $x_i$. In reality, there are various mechanisms to generate boomerang packets that work in different network environments. For instance, if a virtual gateway router of a VM is allowed to forward packets through the interface that the packets came from, the VM can manipulate the layer-3 address of packets to generate boomerang packets without additional virtual interfaces or helper nodes. Consider an ICMP boomerang packet as an example. If we make a layer-3 boomerang packet with ICMP echo request header as follows:

```plaintext
<Egress Probe Packet>
srcMAC:A_MAC dstMAC:GW_MAC
srcIP:A_IP dstIP:A_IP proto:ICMP
type:8 code:0 id:123 seq:355
```

(where each A_MAC, A_IP, and GW_MAC refers to the MAC address of the monitoring VM, the IP address of the monitoring VM, and the MAC address of the gateway), then the gateway will forward this packet back to the source node after it changes MAC addresses as follows:

```plaintext
<Ingress Probe Packet>
srcMAC:GW_MAC dstMAC:A_MAC
srcIP:A_IP dstIP:A_IP proto:ICMP
type:8 code:0 id:123 seq:355
```

This ingress probe packet does not bypass the ingress firewall, because this does not match what the Connection Tracking System (conntrack) [12] expects: the conntrack waits for the corresponding

---

$^3$In a cloud environment, a packet may sometimes legitimately pass through a virtual firewall due to an implicit rule. Since this implicit rule is not visible to cloud users, the packet may seem to bypass the virtual firewall from the cloud user’s perspective. Because we describe this attack from the cloud user’s perspective, we by refer to this as bypassing the firewall, even though this is not what is happening in the strictest sense.
ICMP echo reply. Therefore, as long as we do not send an ICMP echo reply, the boomerang probe packet goes through the ingress-firewall and is processed by the explicit rules; it may or may not be allowed through, depending on the rules that have been configured in the security group. This feature allows one to utilize both the ingress and egress rules for probing the iptables update time. In the rest of this chapter, we explain our work based on the single-interface scenario with the layer-3 ICMP boomerang packets. In Section 4.4.2, we describe more details of practical deployment of this attack.

### 4.4.3 Practical Epoch Monitor

Figure 4.12 shows the architecture of our Epoch monitoring system, named **EpochMonitor**. The **EpochMonitor** is designed as a standalone system that can be applied to various environments including restricted environments (e.g., monitoring with a single VM with a single virtual interface). For a standalone system to monitor its own Epochs, two conditions must be satisfied: 1) the monitoring VM should be able to configure its own virtual firewalls (typically by making API calls to the cloud infrastructure); 2) the monitoring VM should be able to generate probe packets that go

![Figure 4.12: Architecture of EpochMonitor.](image-url)
through the virtual firewall and come back to itself. \(^4\) We discuss more details about the conditions and deployment environment in Section 4.8. For this section, we will simply assume the two conditions are met.

The *EpochMonitor* consists of five components: Epoch manager, request sender, probe sender, probe monitor, and analyzer. The Epoch manager orchestrates other modules to conduct the given Epoch monitoring such as the iterative *UPDATE+PROBE* and the reactive *UPDATE+PROBE* algorithms. The request sender sends firewall rule-updating requests through the cloud API. The probe sender starts to generate a series of corresponding boomerang probes, which can be monitored at the virtual interface of the VM. The probe monitor keeps checking this virtual interface and reports back to the Epoch manager once it found a change of Epoch. Once an epoch change is detected, the Epoch manager immediately starts the next Epoch probing by repeating the previous steps, as we described in the Algorithm 4. The analyzer can be understood as a front end of the *EpochMonitor* that exports processed information about Epochs. Note that, in the iterative *UPDATE+PROBE* method, the request sender and the probe sender do not need to be orchestrated by the Epoch manager once they started. This means we can place these components outside of the system, in case the conditions for the standalone monitoring are not met.

### 4.5 Covert Channels

To show the utility of the firewall-based information leakage channel, we have implemented prototypes of two different covert channels between isolated VMs (i.e., VMs that through cloud mechanisms are supposed to be isolated so that communication is impossible). The first is a classic covert channel between two VMs residing in the same physical machine. The second is a broadcast-style covert channel, where the sender VM sends messages to multiple receiver VMs scattered across a datacenter.

Especially in a cloud environment, since many different role players are involved, there can be various covert channel attack scenarios. One of the most probable scenarios would be a *covert channel attack from an appliance seller*. In cloud appliance markets \([6, 58]\), cloud users can purchase

---

\(^4\) The conditions are required only for the standalone case, and there are various alternative ways to monitor epochs even if the conditions are not met. E.g., if *condition 1* does not hold, the monitoring system should deploy the iterative *UPDATE+PROBE* and there should be another node that has access to the virtual firewall configuration API for the VM and can run the request-sender component of the *EpochMonitor*. If *condition 2* is not met, there should be another node that can send probe packets to the VM. Since these alternative ways also can be implemented in a similar architecture of the *EpochMonitor*, we present the architecture of *EpochMonitor* here as a reference system architecture.
VM images from sellers. Once an appliance user purchases an image, the cloud user can configure and create an actual VM instance based on the image in his own virtual datacenter. Since appliance sellers also want to protect their software in the VMs, it is commonplace for senders to prevent direct access to the VMs through techniques such as allowing only SSH. Under this environment, we can imagine a very probable covert-channel attack scenario: the image seller may try to obtain some sensitive information in the VM based on his image through a covert channel attack. In this situation, since the information is leaked through a covert channel, the user cannot prevent this attack even if he isolates the VM from the public network. In addition, since the user cannot directly access the VM, it is not easy for the user to determine if the VM leaks some sensitive information or not.

4.5.1 Single-Node Covert Channel

4.5.1.1 Threat Model

We assume that the sender and the receiver VMs are co-residing in a physical machine, and the sender VM is completely isolated from the external network so that it cannot directly leak any confidential information through the network. We also assume that the sender VM has access to the cloud API system and it has control over its own firewall rules through the API. Under this setup, the sender and the receiver VMs can make use of a covert channel based on the firewall to send a confidential message to the receiver VM. Figure 4.13(a) illustrates the environmental setup of this threat model.

4.5.1.2 Covert Channel Mechanism

We exploit the property of the shared execution that any operation for one VM can influence on the execution time of operations for another VM for this covert channel. To be more specific,
the sender VM controls the durations of Epochs by adjusting the load of the shared execution. The message is encoded as a stream of bits. When the sender VM wants to send a 1 for the next Epoch, it makes firewall-related requests that will be reflected during the next iptables update event. To send a 0, it simply does nothing. Meanwhile, the receiver VM keeps monitoring the durations of Epochs using any `UPDATE+PROBE` method. In this way, the receiver can read a series of durations of Epochs, which are either short for 0 or long for 1.

Note that the sender VM also needs to monitor the durations of Epochs to time its sending of the firewall change messages. Since the sender needs to take action reactively to the iptables update event, the reactive `UPDATE+PROBE` method is preferred to the iterative `UPDATE+PROBE` for the sender VM.

### 4.5.1.3 Implementation

We implemented a prototype of the single-node covert channel for OpenStack Mitaka in Python. Since both the sender and receiver VMs need to monitor the durations of Epochs, both VMs used the `EpochMonitor` with add/delete-a-rule requests. The sender additionally uses attach/detach-a-group requests to send information on the channel. As we saw in Section 4.3, there can be a `task-postponing` problem if we use add/delete requests together with attach/detach requests. To address this, we have the sender use two Epochs to send each bit: the sender sends the actual bit in the first Epoch and is always idle in the second. Thus, the receiver can get the bit either from the first Epoch or, if the task was postponed, from the second. For the probes, we utilized IMCP echo request boomerang packets with ingress rules as discussed in Section 4.8.

When the receiver notices a change of Epoch, it decodes the duration of the previous Epoch as one bit of the message. The sender reacts to a change of Epoch by sending either an overhead-introducing request or not depending on the next message bit. This is implemented in the message sender module, which receives push-notifications from the `EpochMonitor`.

### 4.5.2 Multinode Covert Channel

#### 4.5.2.1 Threat Model

In contrast to the single-node scenario, we do not assume the sender and the receiver necessarily co-reside for the multinode covert channel. We also do not assume that the sender VM has control over its own firewall rules. Instead, we assume that the sender VM has control over firewall rules for some other VMs scattered across the datacenter. Of course, the sender VM is also assumed to
be isolated from the external network so that it cannot send confidential messages directly to the receivers through the network. In addition, we assume that the receiver VMs are co-residing with one or more of the VMs whose firewalls are controlled by the sender VM. Figure 4.13(b) illustrates this environment.

This scenario is suitable for a case where it is difficult or even impossible for the receiver VM to co-reside with the sender: for instance, a case where the sender VM is deployed as a ‘dedicated instance’. As more cloud users are concerned with co-residency attacks, cloud providers have started to support ‘dedicated instance’ options, where an entire physical machine is dedicated to a specific customer. Since the cost for the dedicated instances is higher, it is reasonable for cloud users to deploy only the VMs that handle sensitive information as dedicated instances. Under this restricted environment, the classic hardware-based covert channels do not work. We will show the sender can leak confidential messages through the firewalls of other VMs that it controls as a covert channel under this environment.

4.5.2.2 Covert Channel Mechanism

Under the given environment, we should first consider the limitation that the sender VM is unable to know whether messages are sent successfully. Since the firewall update requests do not make any changes on the sender’s own VM’s iptables, the sender is unable to “read” the signal on the channel. Moreover, even if the effects of the requests were visible to the sender, it is not meaningful for the sender VM to monitor its own iptables update events because the updates are not synchronized across hosts, meaning that the Epoch start times will be different on the sender and receiver(s). Therefore, it is important for this covert channel to send messages in a noise-tolerant way. In our proof of concept, we simply assume the message can be safely sent if each bit of the message is sent for $n$ seconds repeatedly. A future implementation could use forward error correction for greater efficiency.

To send the same bit repeatedly for $n$ seconds, we could reuse the single-node covert channel mechanism. However, this would require the sender to invoke many API calls when it sends a ‘1’ bit: i.e., the sender would need to make an overhead-introducing request every second for $n$ seconds. Instead, we use a different way to influence the shared execution. In Section 4.3.2, we saw some requests that can permanently increase or decrease the execution time of the iptables update process. Therefore, the sender can exploit these special requests to send either 1 (by “permanently” increasing the execution time) or 0 (by “permanently” decreasing the execution time) for as long as it needs:
API requests are now only necessary on the “edge” between transmitting a ‘0’ bit and a ‘1’ bit or vice versa. The receiver uses the same monitoring mechanism as the previous covert channel.

4.5.2.3 Implementation

As a proof of concept, we implemented this multinode covert channel for OpenStack Mitaka in Python. The sender attaches or detaches a security group with a large number of rules (e.g., 2,000) as its method of “permanently” increasing and decreasing the execution time of iptables updates. The architecture of the receiver is similar to the previous covert channel’s implementation except that it uses the iterative UPDATE+PROBE mechanism and utilizes both the creation and deletion of rules.

4.6 Evaluation

In this section, we demonstrate the feasibility of our firewall-based information leakage channel by presenting practical evaluation results for both types of covert channels, as well as a side channel to eavesdrop on infrastructure-level events. For this evaluation, we deployed an OpenStack Mitaka IaaS cloud consisting of one controller node, one network node and three compute nodes in Emulab [93]. The type of all nodes used for this evaluation was Emulab d710 [87]. The polling_interval was set to be the default value (2 seconds).

4.6.1 Accuracy ofEpochMonitor

We first evaluated the accuracy of EpochMonitor’s estimation for the size of Epochs. We ran two EpochMonitors in different VMs concurrently and had both measure the size of Epochs while we were generating arbitrary requests. We evaluated the accuracy by comparing their results to the ground truth, which is directly collected from the host machine’s firewall agent. The evaluation result showed both VM’s estimation were very precise: across 460 Epochs measured, each VM showed 684 microseconds and 649 microseconds of root mean square error (RMSE). Each VM’s maximum error was 1.54 milliseconds and 25.5 milliseconds, which is sufficient for distinguishing different requests, which typically show more than 100 milliseconds of difference as we saw in Section 4.3.

4.6.2 Single-Node Covert Channel

For this evaluation, we created two co-residing VMs: a sender and a receiver belonging to different tenants. We had both the sender and the receiver run EpochMonitor with the reactive UPDATE+PROBE method. Both VMs used add/delete requests to monitor Epoch lengths. The sender
used attach/detach requests to send messages by manipulating Epoch durations. As discussed in Section 4.5.1, to properly handle the *task-postponing* problem, we used two epochs to send each bit. The sender sent the message “hello world” encoded in ASCII. Figure 4.14 shows the “hello” part of the result.

As one can see from the figure, the message was sent without much noise. With a naive decoding method that simply interprets any Epoch taking longer than 2.2 seconds as a ‘1’ bit, the receiver had a 0% error rate. For a real-world deployment, more robust error-handling strategies would be required since the environment would be noisier. This could easily be accomplished through more distinguishable patterns as we saw in Section 4.3 or through forward error correction.

For the “hello world” message, the actual bandwidth of the covert channel was 0.211 bits/second. Note that, since we sent ‘1’ bits using two epochs, the best bandwidth for this covert channel is 0.25 b/s (in case the sender sends only ‘0’). Though the bandwidth can be further improved by utilizing more patterns and encoding multiple bits in an epoch, we believe the current bandwidth would be enough for some use cases such as co-residency detection or leaking cryptographic keys.

### 4.6.3 Multinode Covert Channel

For the multinode covert channel attack, we used three VMs. In this case, the sender and receiver are on different hosts. The third VM, used as an intermediary, is co-resident with the receiver. As introduced in Section 4.5.2, the sender and intermediary VMs belong to the same tenant, and the sender could update the firewall of the intermediary. The receiver belongs to a different tenant. The receiver ran *EpochMonitor* with the reactive UPDATE+PROBE method and issued add/delete

![Figure 4.14](image_url)  
*Figure 4.14: Execution durations of the RPC loop while sending “hello” through the single-node covert channel.*
requests, and the sender sent the string “hello world” through the intermediary using long-term impact requests, attachment and detachment of a security group with 2,500 rules. The sender used 10 seconds to send each bit. Figure 4.15 shows the “hello” part of the message as observed by the receiver.

One can see from the results that the receiver saw the message very clearly. The bandwidth was only 0.1 b/s since each bit was sent for 10 seconds. There were several iterations (for example, the 107th and 118th) in which no firewall-updating requests arrived. This is because the receiver used the reactive UPDATE+PROBE method. This could be “fixed” by deploying the iterative or n-reactive UPDATE+PROBE methods.

### 4.6.4 Side Channel Attack

We have found that infrastructure-level events — such as VM creation and deletion — also leave their mark on Epoch times and can be used as a side-channel to detect these events. To demonstrate this, we created a single attacker VM that monitors the Epochs using EpochMonitor with the Reactive UPDATE+PROBE method and add/delete requests. Then, we repeatedly created a VM co-resident on the same host as the attacker and terminated it after 10 seconds. Each of the newly created VMs was attached to an existing security group that had ten rules. Figure 4.16 shows the results. Since VM creation and termination involve virtual-network-level changes, as one can see from the results, the signal for these events appears very clearly in the Epochs. For the same reason, one can also see that one may estimate ‘the number of virtual interfaces of created/terminated VM’ through this channel. Since cloud providers typically limit the number of virtual interfaces for a VM by its flavor (i.e., the

![Figure 4.15: Execution durations of the RPC loop while sending “hello” through the multinode covert channel.](image-url)
Figure 4.16: Execution durations of the RPC loop iterations while a VM is monitoring the infrastructure level activities using EpochMonitor. We repeatedly created a VM and terminated it after 10 seconds. We also varied the number of virtual interfaces of the VM as noted in the figure.

larger the VM is, the more virtual interfaces it may have), this information could be used as a good indicator to estimate the size of the VM as well.

If the attacking VM keeps monitoring such infrastructure-level activities through this side channel and builds up profiles for different requests, we believe that it would be possible to extract further infrastructure-level information such as “the number of VMs the host is running” and “the sizes of security groups attached to this host.” This can provide valuable information about the cloud provider itself and can potentially be used to improve other attack vectors. This would represent a meaningful new class of side-channel attacks targeting infrastructure-level information.

4.7 Mitigation Techniques

In this section, we discuss approaches to mitigating firewall-based information leakage channels.

4.7.1 Adjusting the Polling Interval

A simple way to reduce the potential for information leakage through virtual firewall setup is to increase the polling intervals. For example, if one sets the polling interval to 10 seconds, since it is very rare for actual network level updates to take longer than 10 seconds (as we saw in Section 4.3), the loop will complete within 10 seconds most of the time, making it hard for attackers to distinguish difference in epochs between different requests. Though this approach is readily available and may suppress the attack to some degree, there are two problems with this approach. First, this approach does not prevent the attacker from sending/receiving signals sent through the elapsed time before updating iptables. Second, this approach may increase either response time of the request or
the chance for virtual resources to be functionally inconsistent [16]: different parts of the tenant’s infrastructure could see inconsistent resource states for extended periods of time.

Setting the polling interval to be very short is not an effective mitigation strategy. Though it would seem that this prevents multiple requests from being processed within the same iteration, intensive requests (such as attach/detach requests) will still produce noticeable delays.

4.7.2 Request Rate Limiting

Since rate-limiting is a general strategy to suppress DoS-style attacks targeting API frontends, one may consider using rate-limiting to suppress firewall-based information leakage as well. However, compared to DoS-style attacks, the actual request rate needed for these channels is very low. For running EpochMonitor with the Reactive UPDATE+PROBE method, we need just 2 requests per polling interval. We believe more involved analysis on the request rates of tenants is required to detect this type of attack at the service level. One of the possible approaches in this regard is to devise a policy that may effectively throttle requests from these attacks and enforce the policy by extending the existing distributed resource management systems [54, 85].

4.8 Deployment

4.8.1 Deployment Environment

4.8.1.1 With Helper Nodes

The simplest way to probe changes in iptables is to have another VM (say, a helper node) send (or receive) the probe packets. For example, to probe an egress rule $x_i$, the VM sends probe packets to the helper node, which replies once it receives any probe packet for $x_i$. The problem with this approach is that it requires one more VM. Though this might be a trivial problem in cases where the attacker has its own dedicated resources, it can be a challenging problem in some other cases where the attacker uses a compromised node in an isolated environment.

4.8.1.2 Without Helper Nodes

We now describe how to construct “boomerang packets” to remove the need for a helper node. There are several mechanisms by which this can be accomplished, and different ones work in different network environments. In the following paragraphs, we discuss different environmental setups (from the simplest to the most complicated) and suitable mechanisms under each setup.

• Multiple interfaces – Layer-2 boomerang: The simplest (and least common) environment is
the case in which the VM has multiple virtual interfaces connected to the same virtual network (i.e., the same virtual switch). In this environment, the VM can generate a boomerang packet by simply setting the source layer-2 (MAC) address to one interface’s MAC and the destination to the MAC address of the other interface. The switch will simply forward packets from one interface to the other.

- **Multiple interfaces – Layer-3 boomerang:** If the two virtual interfaces are connected to different networks, but there is routing between those networks, the VM can apply a similar approach. Instead of setting the MAC addresses, it can set the source and destination layer-3 (IP) addresses of the boomerang packet.

- **Single interface – Layer-2 boomerang:** If the previously described conditions are not met, the attacker may try similar techniques with a single interface if the network environment satisfies some other conditions. For example, the layer-2 address manipulation technique may work if the connected virtual switch does not drop packets with the same source and destination MAC addresses. However, we have found this to be unlikely in most production clouds, since it is one of the most primitive features of layer-2 switch devices to maintain the MAC address table and forward/drop packets based on the table. According to our test results, this layer-2 boomerang packet is silently dropped by virtual switches in OpenStack Icehouse (released in 2014) and Mitaka (released in 2016).

*Single interface – Layer-3 boomerang:* As with the case of the layer-2 boomerang, if the gateway router is allowed to forward packets through the interface that the packets came from, the layer-3 address manipulation technique works with a single interface. To be specific, if the attacker VM sends its gateway router a fabricated packet whose source and destination IP addresses are its own but with the destination MAC address of the gateway, the gateway router will naturally forward the packet back to the destination, the VM.

In contrast to layer-2 switch devices, this ‘U-turn’ forwarding is commonplace for routers. In a network, even if the source node does not know the correct route to a destination, its gateway router is in charge of forwarding its packet through a proper route; therefore, most commercial routers are configured to forward a packet back through its ingress port in case its routing table is indicating the port. Though some routers may also send an ICMP redirect packet to the source node to ‘recommend’ that it use the better route, the routers still forward the original packet back to its destination: i.e., the VM.

There can of course be commercial routers that are manually configured to drop packets with the same source and destination IP addresses. However, we argue that it is unlikely to be found in
our target environment because (1) in a current cloud environment, the virtual routers are typically
less feature-rich than commercial routers, and (2) it is not a common practice for operators to block
this type of boomerang packets if they are not generated aggressively. We have tested the behavior
of virtual routers in two different versions of OpenStack, Icehouse (released in 2014) and Mitaka
(released in 2016). In both versions, the virtual routers were not prevented from forwarding packets
destined to the source, and they were also configured to send ICMP redirect packets. In addition, in
both versions, there were no configurable options related to this issue not only for cloud users but
also for cloud providers. In the following section, we explain details about protocols and rules for
the single-interface layer-3 boomerang scenario.

4.8.2 Protocols and Rules

When we make a firewall rule to allow a certain type of connection, it is natural to allow both
request packets in one direction and its counterpart responses in the opposite direction. For instance,
if one makes a rule to allow SSH connections from an external terminal node (say A) to a VM in a
cloud (say B), one must allow not only TCP traffic from A to B’s port 22, but also TCP traffic from
B’s port 22 to A’s port used for the connection. Should one always make a pair of (or more) rules to
allow a type of connection? Fortunately, the answer is No since security group rules in most of the
cloud platforms are stateful [10, 5, 64].

In OpenStack, the statefulness of security group rules is enabled by the Connection Tracking
System (contrack) [12]. Briefly speaking, contrack is a module that estimates the current states of
network connections by inspecting the header of each packet. This connection state information can
help iptables to dynamically filter the packets related to existing connections. In the previous SSH
example, if one sets a firewall rule for A to B’s port 22 and if one sends a SSH request from A’s port
56789 to B’s port 22, contrack will tell iptables that packets from B’s port 22 to A’s port 56789 are
related to an existing connection. Iptables will then implicitly let the egress packets pass through the
firewall.

For monitoring infrastructure-level activities through firewalls, a problem with contrack is that
it may or may not let boomerang packets work: probe packets may be silently dropped at the firewall
or bypass the firewall rules, depending on protocols, rules and system versions. Therefore, the
attacker must be careful when he chooses protocols and rules for the side channel. In this section, we
introduce several representative protocols and rules for generating boomerang packets.
4.8.2.1 ICMP Ping

Before we present the details of boomerang packets and conntrack, we start with the simpler protocol: ICMP ping.

Assume an attacker’s VM (say A) has a reachable node (say B) in the network, and the node B replies to ICMP echo requests. Under this setup, the attacker VM A can use ICMP echo request/reply as probe packets for a firewall rule:

Allow Egress ICMP type:8 code:0 dst:B_IP

To probe if this firewall rule is established, the attacker VM can start to ping the node B with probe packets as follows:

srcIP:A_IP dstIP:B_IP proto:ICMP
type:8 code:0 id:123 seq:0-65535

If a probe packet successfully passes through the firewall rule, it will arrive at node B, and the node will send back a corresponding echo reply packet with the same id and sequence number. This echo reply packet can also successfully pass through the firewall because conntrack makes a special ingress rule for it when it first sees the counterpart echo request packet. Therefore, the attacker node can estimate the time when the firewall rule is established by checking the departure time of the echo request packet which corresponds to the first-arrived echo reply packet.

However, this approach may not work if the attacker node reuses the same rule. Once conntrack observes a ping request packet, it creates a new connection entry for the ping based on the following five tuple of the packet: source IP, destination IP, ICMP type, ICMP code, and ICMP ID. From this point, any packets that have the same tuples value bypass the firewall until the connection entry expires. This means, in the previous example, except for the first ping request packet, every ping request packet generated by the ping process will bypass the firewall, even after the rule is deleted.

For this reason, the attacker cannot simply use the vanilla ping program, which does not support an option to change ICMP code and ID values. Alternatively, if the attacker can directly generate ICMP echo request packets, the attacker can use ICMP rules and probe packets as follows:

<Rule x>
Allow Egress ICMP type:8 code:k

---

5To be more precise, iptables will check the echo reply packet against a special rule, and the rule will query conntrack to determine if the packet is related or belongs to any existing connections.
where \( k \in \{0, \ldots, 255\} \). Here, there exist 256 different ICMP echo request rules (differentiated by the code). Also, each rule has 65,536 different matching connection entries (differentiated by the ID), each of which has 65,536 uniquely identifiable packets (differentiated by the sequence number). This means that the attacker can reuse the same rule and still avoid egress-firewall bypassing by using different ID values. In practice, this number of rules is enough to continuously monitor iptables update events. For example, with a default conntrack and OpenStack setup, the time-out duration of ICMP connection entry is 30 seconds\(^6\) and OpenStack’s iptables update period is 2 seconds. Therefore, the minimum number of available echo request rules for the n-UPDATE+PROBE method is \( 15 \times n \). This number further decreases to \( \frac{15 \times n}{65535} \) if one reuses the same rule with different probe IDs.

The egress-firewall bypassing problem may or may not happen depending on the specific cloud platform and its configuration. For example, in earlier versions of OpenStack, we can observe the egress-firewall bypassing problem. Likewise, according to the Amazon AWS user guide\(^7\), AWS security groups also have the same firewall-bypassing phenomenon. However, in newer versions of OpenStack, the same problem does not occur because the connection entries in the conntrack are explicitly terminated when their corresponding firewall rules are deleted\(^6\).

A limitation of this approach is that we cannot use ingress rules for probing; since ICMP request packets make corresponding reply packets bypass the firewall, ingress rules matching these ICMP echo reply packets can never be probed through this mechanism. This can be a serious limitation depending on the situation of the attacker VM, which we will discuss in Section 4.8.2.6.

### 4.8.2.2 ICMP Boomerang Packets

The aforementioned limitation of ICMP ping as a probing mechanism arises fundamentally because the conntrack module expects the ingress probe packet to come when it first sees the

---

\(^6\)The connection entry expires if no related packet comes during the time-out duration.

\(^7\)Since cloud providers do not reveal their infrastructure-level details, it is difficult to precisely understand the internal connection-tracking mechanism of each cloud provider. However, for Amazon AWS, the description about behavior of their security group’s connection tracking mechanism is exactly the same as that of the Linux conntrack-tool. Thus, we may guess that AWS utilizes a similar iptables and connection-tracking system to implement its security group system.
egress probe packet. This means that if an ingress probe packet is seemingly unrelated to its egress counterpart, the pair of packets can be utilized for probing regardless of the behavior of the conntrack module.

Fortunately, one can make use of the previously introduced layer-3 boomerang mechanism to make probe packets with this property. For example, if one makes a layer-3 boomerang packet with an ICMP echo request header as follows:

```plaintext
<Egress Probe Packet>
srcMAC:A_MAC dstMAC:GW_MAC
srcIP:A_IP dstIP:A_IP proto:ICMP
type:8 code:0 id:123 seq:355
```

(where GW_MAC refers to the MAC address of the gateway), then the gateway will forward this packet back to A after it changes MAC addresses as follows:

```plaintext
<Ingress Probe Packet>
srcMAC:GW_MAC dstMAC:A_MAC
srcIP:A_IP dstIP:A_IP proto:ICMP
type:8 code:0 id:123 seq:355
```

In contrast to the case of ICMP ping, this ingress probe packet does not bypass the ingress firewall because it does not match what conntrack expects — conntrack waits for the corresponding ICMP echo reply. Therefore, as long as one does not send ICMP echo replies, the boomerang probe packet goes through the ingress-firewall and may or may not be filtered depending on the status of the ingress firewall rule. This feature allows one to utilize both the ingress and egress rules for probing the iptables update time.

However, the result may vary depending on ICMP type. For request-type ICMP packets (i.e., type ∈ {8, 13, 15, 17}), conntrack initializes a new connection once it observes an egress packet and waits for the corresponding ingress packet as we described above. For other ICMP types, however, the probe packets are recognized as neither initializing a new connection nor related to any existing connection. Thus, conntrack marks these packets as INVALID. A problem here is that the handling of INVALID packets may vary depending on cloud platform. In older versions of OpenStack, every security group has an implicit rule with highest priority that drops any INVALID packets. This rule prevents any INVALID packet from passing the firewall regardless of whether
there exists a matching rule or not. This has been corrected in newer versions of OpenStack, where other firewall rules have higher priority, so that INVALID packets may also pass the firewall if there exist a matching rule. Figure 4.17 shows snapshots of iptables in different OpenStack versions where the priority of the rule to drop INVALID packet is different. For this reason, non-request-type ICMP cannot be used for probe packets in older versions of OpenStack.

4.8.2.3 TCP Boomerang Packets

For TCP packets, since conntrack initializes a new connection only if it sees a TCP SYN or ACK packet, the attacker may generate TCP SYN and ACK boomerang packets in a similar manner to request-type ICMP boomerang packets. For example, for TCP rule:

```
<Rule x>
  Allow Egress TCP dport:k dst:A_IP
```

the attacker makes use of the following sets of probe packets:

```
<Probes for the rule x>
  srcIP:A_IP dstIP:A_IP
  proto:TCP flags:SYN or ACK
  sport:0-65535 dport:k seq:0-4294967295
```

(a) Iptables chain of OpenStack Juno

(b) Iptables chain of OpenStack Mitaka

**Figure 4.17**: Snapshot of example Iptables chains of different OpenStack. The rule to drop INVALID packets is placed at the top of the iptables chain in OpenStack Juno, but at the bottom in OpenStack Mitaka.
Other TCP packets (including TCP SYN/ACK) are treated as INVALID, similar to non-request-type ICMP packets, so they can be used as boomerang packets for newer versions of OpenStack.

Compared to ICMP, a benefit of the TCP boomerang mechanism is that it has a larger number of rules and connections available. Since TCP firewall rules are differentiated by the destination port number and connections are differentiated by the source and destination port numbers, there can be at most 65,536 different rules and each rule can have at most 65,536 different connections. However, if one does not need a large number of rules and connections, the TCP boomerang mechanism can have a disadvantage because the time-out duration for TCP connection entries is generally longer than ICMP (120 seconds for TCP SYN and 300 seconds for TCP ACK).

4.8.2.4 UDP Boomerang Packets

The behavior of conntrack for UDP packets is very similar to that for request-type ICMP packets: conntrack initiates a new connection when it sees an UDP packet and waits for reply packets. Since UDP does not have any specific form of reply, the reply packets are simply the packets with swapped IP addresses and port numbers. Therefore, for UDP firewall rule:

```plaintext
<Rule x>
Allow Egress UDP dport:k dst:A_IP
```

the attacker can probe the iptables update time using probe packets as follows:

```plaintext
<Probes for the rule x>
srcIP:A_IP dstIP:A_IP proto:UDP
sport:0-65535 dport:k seq:0-4294967295
```

Since the ingress and egress probe packets have the same port numbers, conntrack does not treat the ingress probe packets as the reply for the egress probes, and naturally the ingress probe packets do not bypass the firewall. However, there is an exception: if the source port and the destination port of a boomerang packet are the same, the ingress probe packet will be recognized as a response to the egress probe packet by conntrack. In this case, the ingress will bypass the firewall as in the case of ICMP ping.

4.8.2.5 Other Protocols

The Linux conntrack-tool supports TCP, UDP, and ICMP, and takes a default behavior for other protocols: initiate a connection once it sees a packet, and wait for reply packets in the
opposite direction. Here, a connection is differentiated only by a three-tuple: protocol number and source/destination IP addresses. Therefore, if one creates a boomerang packet with a random protocol number, the ingress probe packet is always recognized as a reply to the egress probe. This is almost identical to the case of ICMP ping and UDP with the same port numbers. However, for these protocols, the default connection time-out duration is much longer (600 seconds) and the numbers of available rules and connections are fewer (253 rules and 1 connection per rule), so these protocols are less desirable for use as boomerang packets.

Table 4.1 summarizes the properties of aforementioned probing mechanisms in two different OpenStack versions.

### 4.8.2.6 Ingress Rules or Egress Rules?

When a rule is updated in iptables, the corresponding probe packets start to pass the firewall and arrive back at the attacker VM. Therefore, one can measure the arrival time of the first-arrived

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Rule Direction</th>
<th>Juno</th>
<th>Mitaka</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICMP Echo Request-Reply</td>
<td>Egress</td>
<td>EB</td>
<td>OK</td>
<td>Default connection time-out: 30 sec. No ingress rule is needed.</td>
</tr>
<tr>
<td></td>
<td>Ingress</td>
<td>IB</td>
<td>IB</td>
<td></td>
</tr>
<tr>
<td>Request-type ICMP Boomerang</td>
<td>Egress</td>
<td>EB</td>
<td>OK</td>
<td>Default connection time-out: 30 sec.</td>
</tr>
<tr>
<td></td>
<td>Ingress</td>
<td>EB</td>
<td>OK</td>
<td></td>
</tr>
<tr>
<td>Non-request-type ICMP Boomerang</td>
<td>Egress</td>
<td>D</td>
<td>OK</td>
<td>Does not make any connection entry.</td>
</tr>
<tr>
<td></td>
<td>Ingress</td>
<td>D</td>
<td>OK</td>
<td></td>
</tr>
<tr>
<td>TCP SYN or ACK Boomerang</td>
<td>Egress</td>
<td>EB</td>
<td>OK</td>
<td>Default connection time-out: 120 sec (SYN), 300 sec (ACK).</td>
</tr>
<tr>
<td></td>
<td>Ingress</td>
<td>EB</td>
<td>OK</td>
<td></td>
</tr>
<tr>
<td>Other TCP Boomerang</td>
<td>Egress</td>
<td>D</td>
<td>OK</td>
<td>Does not make any connection entry.</td>
</tr>
<tr>
<td></td>
<td>Ingress</td>
<td>D</td>
<td>OK</td>
<td></td>
</tr>
<tr>
<td>UDP Boomerang (sport ≠ dport)</td>
<td>Egress</td>
<td>EB</td>
<td>OK</td>
<td>Default connection time-out: 30 sec.</td>
</tr>
<tr>
<td></td>
<td>Ingress</td>
<td>EB</td>
<td>OK</td>
<td></td>
</tr>
<tr>
<td>UDP Boomerang (sport = dport)</td>
<td>Egress</td>
<td>EB</td>
<td>OK</td>
<td>Default connection time-out: 120 sec. No ingress rule is needed.</td>
</tr>
<tr>
<td></td>
<td>Ingress</td>
<td>IB</td>
<td>IB</td>
<td></td>
</tr>
<tr>
<td>Boomerang using Other Protocols</td>
<td>Egress</td>
<td>EB</td>
<td>OK</td>
<td>Default connection time-out: 600 sec. No ingress rule is needed.</td>
</tr>
<tr>
<td></td>
<td>Ingress</td>
<td>IB</td>
<td>IB</td>
<td></td>
</tr>
</tbody>
</table>
ingress probe packet and estimate the time when the iptables are updated. At this point, there is a difference between ingress rules and egress rules. If an ingress rule is used, the arrival time of the first passed probe packet is almost identical to the iptables update time, and the attacker VM can immediately notice the event. However, for egress rules, once the iptables are updated, the first passed probe packet must make a round trip before it arrives back to the attacker VM. In this case, the actual iptables update time will be close to the departure time of the probe packet, so the VM should also search for the departure time of the matching probe packet. Naturally, there will be longer delay between the notification time and the actual event time in case where egress rules are utilized. Therefore, in cases when the attacker needs immediate notification of the iptables update event (like the Reactive UPDATE+PROBE method), utilizing ingress rules is preferred. In addition, since it is commonplace for cloud users to set an egress rule that allows any traffic, utilizing egress rules may not be an available option in all cases.

However, there is one important benefit of utilizing egress rules. Since egress rules drop probe packets just after the probe packets depart, most of the probe packets will not go through the cloud provider’s network and the total amount of the probe traffic can be significantly decreased. This property can be especially helpful in decreasing the chance of being detected by the cloud administrator’s network monitoring systems.

4.8.2.7 Probing Deletion of Rules

So far, we have estimated the update time of iptables by measuring the time when the creation of a firewall rule is actually reflected on the iptables. This is because noticing the creation of rules is more immediate and definitive than deletion. When a rule is created, a probe packet that passes the firewall will be directly observed by the attacker VM. However, when deleted, the VM will simply observe there is a long delay in the arrival of the probe packets, since the last passed probe packet has arrived, so that it cannot judge whether the delay is due to the update of the iptables or due to network congestion. Therefore, it is necessary to conduct statistical analysis on the arrival and departure times of probe packets to notice iptables update times due to the deletion of a rule, and this can be a serious obstacle for reactive systems.

Nevertheless, using rule deletion time can be useful in cases where the number of available rules is very limited and the immediate notification of the iptables update event is not required (as in the case of the Iterative UPDATE+PROBE method).
4.9 Related Work

4.9.1 Information Leakage Channels in Cloud

Ristenpart et al. [77] first introduced several information leakage channels and described their potential impacts in the cloud. Specifically, the authors utilized covert channels based on hard disks and caches to verify the co-residency of two cooperative VMs, achieving a bandwidth of 0.2 b/s. The performance of this covert channel was further improved by Xu et al. [97] to the extend of 233 b/s. To overcome the limitations of cache-based covert channels, Wu et al. exploited the memory bus as a covert channel mechanism and achieved a transfer rate of 190 Kb/s. Liu et al. made a great improvement to the bandwidth of the cache-based covert channel by adopting the PRIME+PROBE technique [70], achieving 1.2Mb/s [53] in situations where two VMs are hosted on the same processor package. Bates et al. [18] exploited the physical network interface as a side channel to detect co-residency; in their scenario, the probing VM could detect co-residency with the victim by monitoring the network performance change of the victim. A common factor among these previous studies is that they focus on hardware-level shared resources, while our work focuses on software-level cloud management services.

In addition to the aforementioned work, Ristenpart et al. [77] exploited additional side channels based on the behavior of a cloud management system, including host machine’s IP addresses, interVM network round-trip time, and the numerical distance of internal IP addresses to detect co-residency as well as the VM placement policy of a cloud. Varadarajan et al. [89] and Xu et al. [99] showed that the previous approaches do not work in modern cloud platforms, but that there still exist several factors that may increase the probability of co-residency. In the sense of analyzing and utilizing the property of cloud management systems, these studies are the most similar to our work. However, since our side channel is based on the fundamental software architecture of the management system (i.e., shared processes), it would be more difficult to suppress this type of sidechannel.

4.9.2 Shared Services and Distributed Resource Management

Although there has not been a study exploring the potential threat due to the exploitation of shared services in a multitenant environment, there have been related studies exploring problems in shared services in terms of distributed resource management. Mace et al. [54] grouped related requests into a ‘workflow’ (e.g., requests from a certain tenant) and suggested controlling the process rates of workflows through a centralized framework. That framework, called Retro, can monitor
the resource usage of each workflow from distributed shared services and enforce policies on them. For the same problem, Suresh et al. [85] introduced the Wisp framework, which can allow localized workflow-process rate control at each individual service. Since these frameworks were not originally designed to mitigate the information leakage attacks introduced in this chapter, they may not be suitable to detect and mitigate our attacks that have relatively low request rate. However, in terms of monitoring requests and controlling their process rates, one may consider extending these prior studies to build a protection system against the attacks.

4.10 Conclusion and Future Work

In this chapter, we have shown that the shared network management architecture in the cloud can be exploited to build an information leakage channel. Through our evaluation, we have demonstrated that we can build robust and unique information leakage channels by exploiting the virtual firewall infrastructure. As future work, we plan to refine the analysis technique to extract more precise information from the firewall-based side channel, allowing for more accurate detection and higher bit rates.
CHAPTER 5

CONCLUSION

5.1 Summary of the Dissertation

The multiparty, multilayer environment of public clouds inherently limits the visibility of both cloud providers and cloud tenants into the whole system, and the limited visibility complicates problems involving multiple parties in a cloud. In existing cloud platforms, solutions for these problems often require time-consuming and expensive interaction between different parties. Given that resource optimization and cost saving are becoming top priorities not only for cloud providers but also for tenants, resolving this lack of visibility is becoming a critical challenge for cloud computing platforms.

In this dissertation, we have shown that interparty visibility in a cloud platform can be enhanced, by presenting three different approaches, either in a symbiotic way to cooperatively address cloud problems or in an uncooperative way to exploit it. The CloudSight framework [16] showed a synergistic way to improve a cloud tenant’s visibility into the infrastructure for the benefit of both cloud providers and tenants. The visibility problem from the cloud provider’s side has been addressed through Polygravity [13]. In the last part of the dissertation, through the firewall-based side channel, we have shown an uncooperative and unintended way to improve the visibility of tenants into a cloud provider’s infrastructure.

In Chapter 2, we presented our work on CloudSight in which cloud providers allow tenants greater system-wide visibility through a transparency-as-a-service abstraction. CloudSight collects data from a number of vantage points in a cloud platform, collates the data in a graph database using a resource-centric model, and applies clustering mechanisms to find relationships and dependencies between different data sources and to reduce the domain knowledge required to interpret the data. The resulting resource graph can be readily projected onto a specific tenant plane, to limit a tenant’s view to data associated with his cloud resources, or onto a specific time plane, to enable time-related analysis of cloud events. We presented the design and implementation of CloudSight in the OpenStack cloud platform. We showed the utility of our approach by developing a number applications that
make use of the CloudSight abstraction and use the applications to explore real cloud problems.

In Chapter 3, we proposed Polygravity to determine tenant traffic usage via lightweight measurements in multitenant datacenters. We adopted a tomogravity model, widely used in ISP networks, and adapted it to a multitenant datacenter environment. By integrating datacenter-specific domain knowledge, sampling-based partial estimation, and gravity-based internal sinks/sources estimation, Polygravity addresses two key challenges for adapting tomogravity to a datacenter environment: sparse traffic matrices and internal traffic sinks/sources. We conducted an extensive evaluation of our approach using realistic datacenter workloads. The results showed that Polygravity can determine tenant IP flow usage with less than 1% average relative error for tenants with fine-grained domain knowledge. In addition, for tenants with coarse-grained domain knowledge and with partial host-based sampling, Polygravity can reduce the relative error of sampling-based estimation by $\frac{1}{3}$.

In Chapter 4, we analyzed the exploitability of the cloud network management stack (which is shared among cloud tenants), and introduced a practical method for building information leakage channels by monitoring workloads on the cloud network management stack through the virtual firewall. We demonstrated the feasibility of this attack by implementing a side channel to eavesdrop on infrastructure-level events and two different covert channels in OpenStack.

### 5.2 Future Research Directions

In this dissertation, we presented representative problems due to the lack of visibility in a cloud and various approaches to address them. In this section, we briefly introduce several future research directions that can be explored.

#### 5.2.1 Cooperative Cloud Troubleshooting

As we have discussed in this dissertation, the limited interparty visibility in a cloud brings various challenges to both cloud providers and cloud tenants. As the cloud market continues to become bigger and more competitive, cloud providers will actively release new features to support better visibility in order to attract cloud customers [8, 7, 40, 59]. However, when it comes to cloud security and troubleshooting problems, existing clouds still require human operators’ interventions on both the cloud provider’s side and the tenant’s side, which may take several hours to days for taking an emergency measure, finding the root cause, and fixing the problem. This may be a serious problem for both cloud tenants and cloud providers. For the tenants, it can cause certain parts of their services
to lose availability for hours to days; for the providers, the root cause of the problem may disappear in a short time, so they may even lose a chance to conduct a postmortem analysis of the problem.

As a holistic solution to this problem, we envision a cloud platform supporting fully automated interparty troubleshooting where cloud providers and tenants can cooperate to quickly resolve complex issues in an automated and interactive way. To give an example scenario, when a VM belonging to one tenant experiences excessively low network performance, the monitoring system of the VM may detect the issue and automatically ask the cloud provider to take safety measures (e.g., dedicated network bandwidth or live migration) based on the configuration predefined by the tenant through a “cloud troubleshooting API.” When the cloud provider’s diagnosis system receives the request through the API, it may first quickly analyze the legitimacy of the request by checking the network metrics around the VM, take an appropriate emergency measure (e.g., migration), and then initiate investigation to find the root cause of the issue by collecting packet headers of the co-residing VMs. The diagnosis system may find that several co-residing VMs are aggressively generating network traffic and the traffic pattern matches the traffic pattern of a sampled malware. For further investigation, the diagnosis system may request approval for VMI into the VMs from their owners through a “cloud compliance API” that also specifies the purpose of the investigation, the analysis method, the data protection mechanism, the availability of the result, etc. Upon the approval from the tenants, the system may conduct VMI-based analysis on the VMs and find those VMs are compromised by an attacker. Later, a human security analyst may conduct additional postmortem analysis on these data and help the owners of the VMs to clean out the malware or use the compromised VMs as honeypots.

As shown in the example, for this cooperative cloud platform, there are four key functions that should be supported by the cloud provider’s diagnosis system: the *legitimacy verifier* for the user troubleshooting request, the *high-level root cause analyzer* using infrastructure level metrics, the *cloud compliance API*, and the *low-level root cause analyzer*. Though there is no single system that can support these functions, one may use several existing works as a reference to build this system. The *legitimacy verifier* could be built based on the existing cloud metric monitoring tools [8, 40, 59, 65]. The *high-level root cause analyzer* may reuse existing cloud metric analysis tools such as ATOM [29]. The *cloud compliance API* does not have exactly the same system, but Access Transparency [39] of Google cloud could be a closely related reference for this. The *low-level root cause analyzer* can be built based on various low level analysis tools and projects.
[30, 36, 71, 45, 84, 14, 52]. In addition, CloudSight [16] can be put before the cloud troubleshooting API to help cloud users prescreen the problems that are due to their own fault.

5.2.2 Polygravity for Very Large Scale Datacenter Network

Although the practical computation time of Polygravity could be within 5 minutes for our simulated datacenter hosting 300 virtual machines, if it is applied to much larger-scale datacenters (e.g., a datacenter with millions of virtual machines), it would not be feasible to simply apply the current Polygravity model. We suggest two possible approaches to improve the scalability of Polygravity. First, by aggregating multiple terminals (i.e., virtual machines) into one entity, one may reduce the problem size and decrease the computation time. This approach could be useful when only a selected set of flows are of interest. For this approach, it is important to examine how well the model may maintain its accuracy for a given arbitrarily grouped distributed terminals. This may also help to determine if Polygravity can allow an analyst to aggregate distributed virtual machines by their tenants/type/role or not.

The second approach is directly improving the performance of singular value decomposition (SVD) for the network routing matrix that is the bottleneck of Polygravity. There could be multiple sub-approaches for this, such as applying approximated SVDs, applying the incremental SVD by reflecting the fact that the majority of routing matrix elements does not change frequently, and customizing SVD by reflecting unique properties of the datacenter traffic matrix. (For example, most of the elements of a network routing matrix are either 0 or 1, and very few have larger values.)

5.2.3 Further Exploitation of Shared Software Resource

In Chapter 4, we presented a proof-of-concept of a firewall-based virtual machine counter that shows that the updated time of the virtual firewall can be exploited as a side channel for infrastructure-level activities in the host. We believe this attack can be extended to monitoring other infrastructure level activities such as the creation/deletion of virtual networks, subnets, ports, or security groups because all these activities are processed by the same RPC loop.

Another important future research topic is to review the whole architecture of the cloud management software and examine if there are any other shared software resources that can potentially leak sensitive information to other tenants. This would involve assessing the security of widely used request-processing modes such as batch processing, parallel processing, and transaction processing.
in an multitenant environment, which may help to bring up a new paradigm for a secure multitenant software architecture.
REFERENCES


