

---

# Automated Security Analysis of Virtualized Infrastructures

---

Zur Erlangung des akademischen Grades Doktor-Ingenieur (Dr.-Ing.)  
genehmigte Dissertation von Sören Bleikertz, M.Sc. aus Leverkusen  
Tag der Einreichung: 15.02.2017, Tag der Prüfung: 17.05.2017  
2017 — Darmstadt — D 17

1. Gutachten: Prof. Dr. Michael Waidner
2. Gutachten: Dr. Thomas Groß, University of Newcastle upon Tyne



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Security in Information Technology  
Fachbereich Informatik

# Automated Security Analysis of Virtualized Infrastructures

Genehmigte Dissertation von Sören Bleikertz, M.Sc. aus Leverkusen

1. Gutachten: Prof. Dr. Michael Waidner
2. Gutachten: Dr. Thomas Groß, University of Newcastle upon Tyne

Tag der Einreichung: 15.02.2017

Tag der Prüfung: 17.05.2017

Darmstadt – D 17

---

# Erklärung zur Dissertation

Hiermit versichere ich, die vorliegende Dissertation ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Horgen, Schweiz, den 16.07.2017

---

(Sören Bleikertz, M.Sc.)



---

# Abstract

Virtualization enables the increasing efficiency and elasticity of modern IT infrastructures, including Infrastructure as a Service. However, the operational complexity of virtualized infrastructures is high, due to their dynamics, multi-tenancy, and size. Misconfigurations and insider attacks carry significant operational and security risks, such as breaches in tenant isolation, which put both the infrastructure provider and tenants at risk.

In this thesis we study the question if it is possible to model and analyze complex, scalable, and dynamic virtualized infrastructures with regard to user-defined security and operational policies in an automated way. We establish a new practical and automated security analysis framework for virtualized infrastructures. First, we propose a novel tool that automatically extracts the configuration of heterogeneous environments and builds up a unified graph model of the configuration and topology. The tool is further extended with a monitoring component and a set of algorithms that translates system changes to graph model changes. The benefits of maintaining such a dynamic model are time reduction for model population and closing the gap for transient security violations.

Our analysis is the first that lifts static information flow analysis to the entire virtualized infrastructure, in order to detect isolation failures between tenants on all resources. The analysis is configurable using customized rules to reflect the different trust assumptions of the users. We apply and evaluate our analysis system on the production infrastructure of a global financial institution. For the information flow analysis of dynamic infrastructures we propose the concept of dynamic rule-based information flow graphs and develop a set of algorithms that maintain such information flow graphs for dynamic system models.

We generalize the analysis of isolation properties and establish a new generic analysis platform for virtualized infrastructures that allows to express a diverse set of security and operational policies in a formal language. The policy requirements are studied in a case-study with a cloud service provider. We are the first to employ a variety of theorem provers and model checkers to verify the state of a virtualized infrastructure against its policies. Additionally, we analyze dynamic behavior such as VM migrations.

For the analysis of dynamic infrastructures we pursue both a reactive as well as a proactive approach. A reactive analysis system is developed that reduces the time between system change and analysis result. The system monitors the infrastructure for changes and employs dynamic information flow graphs to verify, for instance, tenant isolation. For the proactive analysis we propose a new model, the *Operations Transition Model*, which captures the changes of operations in the virtualized infrastructure as graph transformations. We build a novel analysis system using this model that performs automated run-time analysis of operations and also offers change planning. The operations transition model forms the basis for further research in model checking of virtualized infrastructures.



---

# Zusammenfassung

Virtualisierung ermöglicht eine höhere Effizienz und Elastizität von modernen IT Infrastrukturen, einschließlich *Infrastructure as a Service*. Jedoch ist die operationale Komplexität von virtualisierten Infrastrukturen aufgrund ihrer Dynamik, “Multi-Tenancy” und ihrer Größe sehr hoch. Fehlkonfigurationen und Angriffe von Insidern tragen zu erheblichen operationalen und Sicherheitsrisiken bei. Beispielsweise führen Verletzungen in der Tenant-Isolierung zu Risiken sowohl für den Infrastrukturbetreiber als auch für den Nutzer.

In dieser Dissertation untersuchen wir die Frage, ob es möglich ist komplexe, skalierbare und dynamische virtualisierte Umgebungen zu modellieren und hinsichtlich benutzerdefinierter operationaler und sicherheitsrelevanter Richtlinien in einem automatischen Verfahren zu überprüfen. Wir etablieren ein neues praktisches und automatisches Framework für die Sicherheitsanalysen von virtualisierten Infrastrukturen. Zuerst stellen wir ein System vor, welches die Konfiguration von heterogenen Umgebungen automatisch extrahieren kann und ein einheitliches Graphenmodell der Konfiguration und der Topologie aufbaut. Zusätzlich wird das System mit einer Komponente zur Überwachung der Umgebung sowie Algorithmen ausgebaut, welche es erlauben, Änderungen in der Umgebung in Änderungen im Graphenmodell zu übersetzen. Die Vorteile eines solchen dynamischen Modells sind zum einen Zeiteinsparungen im Aufbau des Modells, als auch das Schliessen der Lücke im Erkennen von vorübergehenden Sicherheitsverletzungen. Unsere Analyse ist die erste, welche statische Informationsflussanalyse auf die gesamte virtualisierte Umgebung überträgt, somit können Verletzungen in der Tenant-Isolierung in allen Ressourcen entdeckt werden. Die Analyse ist mittels benutzerdefinierter Regeln konfigurierbar, welche die unterschiedlichen Sicherheitsannahmen der Benutzer widerspiegeln. Wir verwenden und evaluieren unser System in der Produktionsumgebung eines globalen Finanzinstitutes. Im Rahmen der Informationsflussanalyse von dynamischen Infrastrukturen stellen wir das Konzept der dynamischen, regelbasierten Informationsflussgraphen vor und entwickeln Algorithmen, welche Informationsflussgraphen für dynamische Systemmodelle verwalten.

Wir generalisieren die Analyse von Isolationseigenschaften und etablieren eine generische Analyseplattform für virtualisierte Infrastrukturen, welche es erlaubt eine breite Menge von operationalen und sicherheitsrelevanten Richtlinien in einer formalen Sprache auszudrücken. Die Anforderungen an die auszudrückenden Richtlinien werden in einer Fallstudie mit einem Cloud-Provider untersucht. Erstmals wird eine Reihe von etablierten automatischen Theorembeweisern sowie Modellprüfern für die Analyse von virtualisierten Infrastrukturen gegenüber spezifizierten Richtlinien angewendet. Außerdem überprüfen wir dynamisches Verhalten, wie zum Beispiel die Migration von VMs.

Im Falle der Analyse von dynamischen Infrastrukturen verfolgen wir sowohl einen reaktiven als auch einen proaktiven Ansatz. Unser neu entwickeltes reaktives Analysesystem reduziert die Zeit zwischen Systemänderung und Analyseergebnis. Das System überwacht die Infrastruktur auf Änderungen und verwendet einen dynamischen Informationsflussgraphen unter anderem zur Überprüfung von Tenant-Isolierung. Im Rahmen des proaktiven Ansatzes entsteht ein neuartiges Modell, das *Operations Transition Model*, welches durch Operationen verursachte Änderungen in virtualisierten Infrastrukturen mittels Graphtransformationen abbildet. Ein neues auf dem Modell aufbauendes Analysesystem überprüft automatisch Operationen zur Laufzeit und ermöglicht es außerdem, Änderungen in virtualisierten Umgebungen zu planen. Das *Operations Transition Model* bildet die Basis für weitere Forschungen im Bereich der Modellüberprüfung von virtualisierten Infrastrukturen.





---

# Acknowledgments

Many individuals contributed to the success of this thesis. First and foremost I want to thank my advisors Michael Waidner and Thomas Groß. Michael gave me the freedom to pursue my own ideas and research interests, while providing valuable insights when needed. Thomas has been an outstanding collaborator and mentor during the years of my research.

I want to thank the many collaborators with whom I had the pleasure to work with on a variety of research projects. My very good and long-term friends Sven Bugiel and Carsten Vogel. Matthias Schunter has been a great mentor during my internships at IBM and parts of my thesis work. Sebastian Mödersheim provided valuable insights in formal methods.

Furthermore, large parts of this thesis has been done at the IBM Research laboratory in Zurich and I want to thank the colleagues I worked with there. In particular, my managers Mike Osborne and Andreas Wespi who provided me the support and flexibility to work on my research. I shared many PhD experiences and enjoyed technical as well as non-technical discussions with Animesh Trivedi, Anil Kurmus, Nikola Knezevic, and Zoltan Nagy, who all have been great friends ever since.

Finally, I sincerely thank Eva and my family for their continuous support throughout the years of working on this thesis.

*Sören Bleikertz*  
Horgen, Switzerland



---

# Contents

<b>1. Introduction</b>	<b>13</b>
1.1. Motivation	13
1.1.1. Isolation Failures in Multi-Tenant Infrastructures	15
1.1.2. Further Security and Operational Requirements	17
1.1.3. Problem Statement and Summary	17
1.2. Thesis	18
1.3. High-Level Approach and Scope	18
1.4. Contributions	19
1.5. Outline and Organization	22
<b>2. Background and Literature Review</b>	<b>23</b>
2.1. Background	23
2.1.1. Virtualization	23
2.1.2. Cloud Computing	28
2.2. Literature Review	29
2.2.1. Configuration Discovery and Modeling of Networks and Systems	30
2.2.2. Isolation and Information Flow Analysis	31
2.2.3. Formal Languages for Security and Operational Policies	33
2.2.4. Security Analysis of Infrastructure States and Changes	35
<b>3. Automated Information Flow Analysis</b>	<b>39</b>
3.1. Introduction	39
3.1.1. Contributions	39
3.1.2. Applications	40
3.2. A Model for Isolation Analysis	41
3.2.1. Flow Types	41
3.2.2. Modeling Isolation	42
3.3. Isolation Analysis of Virtual Infrastructures	43
3.3.1. Discovery	44
3.3.2. Translation into a Graph Model	44
3.3.3. Coloring through Graph Traversal	45
3.3.4. The Coloring Traversal Rules	45
3.3.5. Case-Study Traversal Rules	46
3.3.6. Detecting Undesired Information Flows	47
3.4. Security Analysis	47
3.4.1. Configuration Discovery and Translation	47
3.4.2. Graph Coloring and Information Flows	50
3.4.3. Correctness of the given Traversal Rules	50
3.4.4. Traversal Rules Coverage	52
3.4.5. Discussion	52
3.5. Implementation	53
3.5.1. Discovery	53
3.5.2. Processing	54

3.6. Case Study: Virtualized Infrastructure Isolation . . . . .	54
3.7. Summary . . . . .	55
<b>4. Virtualization Assurance Language</b>	<b>57</b>
4.1. Introduction . . . . .	57
4.1.1. Contribution . . . . .	57
4.1.2. Outline . . . . .	58
4.2. Virtualized Infrastructure Security Goals . . . . .	58
4.2.1. Operational Correctness . . . . .	58
4.2.2. Failure Resilience . . . . .	59
4.2.3. Isolation . . . . .	60
4.2.4. Case Study: Policy Scope . . . . .	60
4.3. Language Requirements . . . . .	61
4.4. Language Syntax and Semantics . . . . .	61
4.4.1. Terms and Types . . . . .	62
4.4.2. Function Symbols and Dependent Terms . . . . .	63
4.4.3. Facts, State and Conditions . . . . .	64
4.4.4. Goals . . . . .	64
4.4.5. Structured Specifications . . . . .	65
4.4.6. Dual Type System . . . . .	65
4.5. Definition and Specification of Attack States . . . . .	65
4.5.1. Operational Correctness . . . . .	65
4.5.2. Failure Resilience . . . . .	66
4.5.3. Isolation . . . . .	67
4.6. Review of Language Requirements . . . . .	68
4.7. Summary . . . . .	69
<b>5. Automated Verification of Security Policies</b>	<b>71</b>
5.1. Introduction . . . . .	71
5.1.1. Contributions . . . . .	72
5.1.2. Architecture . . . . .	72
5.2. Language Preliminaries . . . . .	74
5.2.1. Rules and Goals . . . . .	74
5.2.2. Horn Clauses . . . . .	74
5.3. Problem Classes . . . . .	75
5.3.1. Local vs. Global . . . . .	75
5.3.2. Positive vs. Negative Attack States . . . . .	75
5.3.3. Static vs. Dynamic . . . . .	76
5.4. Compiling Problem Instances . . . . .	76
5.4.1. Graph Encoding and Refinement . . . . .	76
5.4.2. Strategy Amendment for Nodes Connectivity . . . . .	79
5.4.3. Encoding Static Problems into FOL . . . . .	80
5.5. Model-Checking a Virtualized Infrastructure . . . . .	81
5.5.1. Zone Isolation . . . . .	81
5.5.2. Secure Migration . . . . .	82
5.5.3. Absence of Single Point of Failure . . . . .	84
5.6. Case Study for Zone Isolation . . . . .	86
5.7. Summary . . . . .	88

<b>6. Dynamic Information Flow Graphs</b>	<b>89</b>
6.1. Introduction	89
6.2. Isolation in Virtualized Infrastructure	90
6.3. Constructing an Information Flow Graph using Flow Rules	91
6.3.1. System and Information Flow Models	91
6.3.2. Information Flow Rules	92
6.3.3. Ordering of Information Flow Rules	93
6.3.4. Application of Rules and Construction of Information Flow Model	96
6.3.5. Algorithm Analysis	99
6.3.6. Summary	103
6.4. Fully Dynamic Information Flow Analysis	104
6.4.1. Translating System Model Changes to Information Flow Changes	104
6.4.2. Processing Connectivity Changes	105
6.4.3. Algorithm Analysis	106
6.4.4. Summary	109
6.5. Implementation	109
6.6. Conclusion	111
<b>7. Near Real-Time Detection of Security Failures</b>	<b>113</b>
7.1. Introduction	113
7.2. System and Security Model	114
7.2.1. System Model	114
7.2.2. Threat Model	115
7.3. Design and Implementation	115
7.3.1. Obtaining Infrastructure Change Events	116
7.3.2. From Change Events to Model Updates	117
7.3.3. Differential Information Flow Analysis	121
7.3.4. Specification of Security Policies and Detection of Violations	121
7.4. Performance Evaluation	122
7.4.1. Methodology and Environments	123
7.4.2. Results and Discussion	123
7.5. Security Evaluation	125
7.5.1. Security Analysis	125
7.5.2. Security Discussion	127
7.5.3. Security Testing	128
7.6. Summary	128
<b>8. Proactive Security Analysis of Changes</b>	<b>129</b>
8.1. Introduction	129
8.2. System and Security Model	130
8.3. A Model of Dynamic Virtualized Infrastructures	130
8.3.1. Modeling of Infrastructure Changes	130
8.3.2. Dynamic Information Flow Analysis	135
8.3.3. Infrastructure Policies as Graph Matches	137
8.4. Automated Analysis: Design and Implementation	140
8.4.1. System Architecture	141
8.4.2. Run-time Analysis of Changes	142
8.4.3. Change Plan Analysis	143
8.5. Evaluation	143
8.5.1. Security Analysis	143



8.5.2. Performance Measurements . . . . .	145
8.6. Summary . . . . .	147
<b>9. Conclusions</b>	<b>149</b>
9.1. Research Summary . . . . .	149
9.1.1. Overview and Comparison of Tools and Approaches . . . . .	149
9.1.2. Summary of Research Questions . . . . .	150
9.2. Discussion and Limitations . . . . .	152
9.3. Future Work . . . . .	153
9.4. Conclusions and Commercial Impact . . . . .	154
<b>Bibliography</b>	<b>155</b>
<b>A. Wissenschaftlicher Werdegang</b>	<b>177</b>

---

# 1 Introduction

The use of virtualization in IT infrastructures has seen a remarkable growth over the last decade. Technically, virtualization enables the sharing of physical resources by providing virtual resources on top of them. The economic drivers behind the adoption of virtualization are higher efficiency, due to consolidation, increased utilization, which leads to cost savings, as well as more flexibility and agility. In fact, virtualized IT infrastructures are the foundation of the increasingly popular *Cloud Computing*, a new service delivery model that promises, among other benefits, a rapid provisioning of computing resources. Start-ups, SMEs, large enterprises, and governments all alike adopt virtualization for its benefits. Start-ups mainly leverage virtualization through the cloud computing paradigm where virtual resources are offered by providers such as Amazon [Ama14b], IBM SoftLayer [Sof14], or RackSpace [Rac14]. Large enterprises, on the other hand, tend to adopt virtualization for their internal IT infrastructure. For instance, EMC [EMC10] and Accenture [Acc12] target or already achieved a virtualization rate of over 90% for their internal servers. IBM advises as best practice for modern data centers a rate of 60%+ for virtualized servers and 80-90% for storage virtualization [IBM12]. The US government, one of the largest consumer of IT, uses both cloud computing as well as private virtualized infrastructures for their benefits [Kun10]. Virtualization had and continues to have a major impact on the IT infrastructures of all kinds of organizations, and changes how these infrastructures are operated and managed.

---

## 1.1 Motivation

---

Although virtualization and cloud computing offer many technical and economical benefits, a major inhibitor in their adoption is security [MG09a]. Virtualized infrastructures have introduced new challenges for the security management and operation compared to traditional physical infrastructures. Garfinkel et al. [GR05] have identified the following challenges.

**Scaling:** New virtual machines can be created rapidly, compared to long procurement and setup procedures of physical servers. This results in a virtual machines (VMs) sprawl where the number of VMs can grow at an explosive rate. Existing procedures for the security management and operation of physical servers fail in this new environment and limited automation with manual intervention increases the risks of misconfigurations and security failures.

**Transience:** The lifecycle of VMs is more dynamic compared to physical servers. As quickly as a VM is created, it can also be suspended, shutdown, or destroyed. This behavior results in an environment that is under constant change and where a steady state might not be reached. Static security mechanisms fail in such dynamic environments and may miss transient security failures.

**Mobility:** In addition to the dynamic lifecycle, VMs may also migrate in the infrastructure between different hosts. This VM mobility further contributes to the infrastructure dynamic behavior and challenges existing security mechanisms. In particular, the source and destination hosts may belong to different security perimeters or the migration itself can be attacked [OCJ08].

**Identity:** Methods that determine ownership of a machine in a static way, for instance, based on the MAC address or Ethernet port, are not feasible anymore for virtualized infrastructure, because of dynamic VM creation with random MAC address generation and VM mobility.

In summary, these challenges stem from two properties of virtualized infrastructures: scale and dynamics. Security challenges at the infrastructure level concern the security administrators and operators of such

---

environments. However, not only do the infrastructure operators face new security challenges, but also the owners of virtual machines. Patch management is becoming more difficult due to branching and VM snapshots, as well as due to a diverse set of operating systems and applications that are running in VMs. Snapshots also impact the cryptographic randomness for applications in the VM, because reverting back to a previous snapshot may lead to the same randomness being generated [RY10]. Further secure deletion of data is a challenge as data must be deleted from all snapshots.

Infrastructure clouds, as part of cloud computing, delivers compute, network, and storage resources by relying on virtualization. Therefore, the challenges of virtualized infrastructures remain also for cloud computing, however also new challenges emerged due to its multi-tenancy and the outsourcing approach in case of public clouds [MS10]. In fact, NIST [MG09a] identified the following key issues in the security of cloud computing: trust, multi-tenancy, encryption, and compliance. ENISA [ENI09] conducted a risk assessment that further investigates security challenges in cloud computing. In addition, the Cloud Security Alliance maintains a list of the top ten threats in cloud computing [CSA10, CSA13]. In the following we aggregate, summarize, and discuss a subset of the major risks and threats of both the ENISA and CSA reports.

**Isolation Failure:** Fundamental traits of infrastructure clouds are multi-tenancy and the sharing of resources. However, physical resources were often not designed for sharing and ensuring proper isolation. The resulting side-channels have been exploited for cloud computing environments that allowed to extract both coarse- and fine-grained information, such as a cryptographic key, from another tenant [RTSS09, ZJRR12]. Furthermore, the complexity of the virtualization layer itself, namely the hypervisor, resulted in software vulnerabilities that can be exploited from a virtual machine, resulting in privilege escalation and isolation failures [Orm07]. Finally, a misconfiguration in the virtualization layer of compute, network, or storage may tear down the isolation of tenants. The Cloud Security Alliance [CSA13] states that *“The key is that a single vulnerability or misconfiguration can lead to a compromise across an entire provider’s cloud.”* They propose strong compartmentalization, monitoring for unauthorized changes, and configuration audits as suitable remediation steps.

**Malicious Insider:** High-privileged individuals, such as the system administrators of the cloud provider, pose a significant risk as they can cause damage on a large scale. For example, an administrator can dump the memory content of a tenant’s VM and with that all sensitive information contained therein [RC11]. The insider threat is well known in organizational security, but its impact is amplified in public clouds as many customers converge under a single management domain. According to [Olt13], 36% of the surveyed enterprises state that the growing usage of cloud computing makes the detection and prevention of insider threats more difficult. With regard to the mitigation of insider threats in public cloud environments, the lack of transparency renders an assessment of the provider’s security processes by a cloud customer almost impossible.

**Compliance Risks and Insufficient Transparency:** For certain types of customers it is mandatory that the cloud infrastructure and the provider meets relevant industry standards and regulatory requirements, such as PCI-DSS for the financial industry. However auditability is often limited and the internal configuration and security processes are not assessable. Providers are reluctant to disclose infrastructure details and do not offer mechanisms to monitor the security of the infrastructure. In addition, cloud customers tend to rush too fast in adopting cloud computing without performing sufficient due diligence in terms of risk and security assessment, thereby dealing with unknown risks.

**Data Protection:** Loss, leakage, or breach of data is a major concern of cloud customers. The root causes can be operational failures, such as faulty storage management, side-channel attacks, or application vulnerabilities, e.g., in a multi-tenant database service. System administrators may maliciously or accidentally delete storage volumes of tenants, or a volume is attached to the wrong customer’s VM. From a jurisdictional side in terms of data protection laws, requirements regarding the location of the stored and/or processed data are important to both cloud providers and customers.



---

Other risks also include the loss of governance and vendor lock-in, the abuse of cloud services, as well as the compromise of the management interface and hijacking of accounts. In summary, cloud computing is challenged by a wide variety of security issues ranging from technical isolation failures, malicious insiders, to compliance and jurisdictional challenges. We will further investigate isolation failures with its potential root causes of misconfigurations, insider attacks, and vulnerabilities.

---

### 1.1.1 Isolation Failures in Multi-Tenant Infrastructures

---

Operating multi-tenant infrastructures, where multiple tenants share the same physical resources, is economically crucial for achieving high utilization. However, isolation of tenants is a critical security requirement in multi-tenant virtualized infrastructures. In enterprise deployments, the tenants often represent different business units, security levels, or application life-cycles. In public virtualized infrastructures as offered by a cloud provider the tenants are in fact different customers, including conflicting organizations and potential malicious ones.

We conducted a survey among IBM's top clients in November 2011, in order to investigate their use of virtualized infrastructures and their requirement of isolation. In this survey, 22 representatives from 14 companies participated with a majority of the participants having the role of system administrator or infrastructure architect. The participants had to select a single answer for each question. The survey covered the areas of automation, how and what is isolated, as well as the means and importance of verifying the isolation. In the following we highlight the results of the poll, with percentages given on the number of participants that selected a specific answer.

- *Deployment Automation:* 20% manual, 56% partially automated with manual intervention, 24% fully automated.
- *Isolation Groups:* 71% based on lifecycle (development, test, production), 25% by clients or compliance related, 4% by data content.
- *Isolation Methods:* 28% VLANs and firewalls, 12% physical separation, 4% only firewalls, 56% a combination of physical, VLAN, and firewalls.
- *Isolation Verification:* 46% manual process, 36% no process at all, 18% automated system.
- *Importance of Isolation Verification:* 40% very important, 40% moderately important, 15% less important, 5% not important.

The survey suggests that while isolation is heavily used in virtualized infrastructures, the understanding and verification of isolation properties is a manual process or largely missing, despite its importance. Manual intervention in the deployment processes makes a majority of the infrastructures vulnerable to misconfigurations and increases the risk of insider attacks. Even though a fully automated deployment may lower these risks, it will not entirely solve the problem due to software bugs or misconfigurations in the automation software itself [Ope14a, Ope14b].

The causes for isolation failures in virtualized infrastructures can be manifold. Chen et al. [CPK10] observe that “As we begin to understand problems in isolation, we should also start to put together an understanding of how different issues and threats combine.”. Indeed, misconfigurations due to complex and heterogeneous environments, insider attacks, as well as software and hardware vulnerabilities may contribute to isolation failures.

#### **Misconfigurations**

Configuration errors have been a major cause for the disruption and failure of IT services in the past [Gra85, OGP03, HB09]. Unsurprisingly, virtualized and cloud infrastructure suffer from similar problems. In

---

particular in large public clouds, misconfigurations will have a major impact since potentially hundreds of thousands of customers may be affected [Ama11, Mic12].

In a virtualized infrastructure we can attribute misconfigurations to its heterogeneity and complexity, as well as to its dynamics and scale. Berger et al. [BCP<sup>+</sup>08] observe that the security management of a networked server running a single workload is already a complex task, and the complexity is amplified when multiple workloads of different tenants are considered. In addition, the virtual machines, their lifecycle and resource assignments have to be considered in the overall security management [FS08].

From private communication with practitioners of virtualized infrastructure deployments we learned that misconfigurations of the virtual network, in particular wrong VLAN identifiers, are a major problem. Administrators often do not understand the entire architecture and make wrong selections of VLAN identifiers. Given the importance of isolation provided by VLANs [VMw09], such misconfigurations may lead to significant security failures due to isolation breaches.

### **Malicious Insiders**

The threat of malicious insiders in organizations is not a new phenomena that emerges in cloud computing, but the threat is amplified due to the disappearance of physical boundaries, which makes it hard to define a security perimeter that divides insiders from outsiders [HNB11].

Claycomb and Nicoll studied the threat of insiders in cloud computing [CN12]. They differentiate between three different kind of insiders. The first kind is a rogue administrator that is financially motivated and tries to steal sensitive information, as well as one that tries to sabotage the provider's IT infrastructure to harm the provider's reputation. An administrator may work on different layers that include the hosting company's infrastructure, the virtual machine images, the virtual machine system, and applications. From a cloud provider perspective the rogue administrator of the infrastructure is a pressing one, since they are usually highly privileged and put both the data of its tenants as well as the provider's reputation at risk. The other two kinds are insiders that make use of the cloud to launch an attack, such as a denial of service or password cracking, and insiders of the cloud customer organization, who try to exploit vulnerabilities in the organization's use of cloud computing.

Various attacks have been presented that a rogue cloud provider administrator could mount to violate isolation properties and steal sensitive information. Rocha et al. [RC11] present attacks which dump the memory of a virtual machine, including all sensitive information that are currently contained therein, or inspect the virtual storage volume for data at rest. Further, attacks on the VM mobility have been shown that extract information while the VM is migrating over an insecure network or even manipulating the VM while in transit to place a backdoor [OCJ08, DGCQ13].

A thin line exists between configuration changes that are accidental and ones that are malicious. A malicious cloud administrator may misconfigure the virtual network, attach a wrong storage volume to a VM, or co-locate VMs of different tenants, in order to allow an attacker to steal sensitive information from a cloud consumer. An administrator could also be the subject of a social engineering attack where the administrator unintentionally contribute to an attack.

### **Hardware and Software Vulnerabilities**

Isolation failures may also stem from vulnerabilities in the hardware or software that is used within a virtualized infrastructure. Physical resources are often shared among multiple tenants, although they were never designed to be shared among mutually untrusted parties. The result are side-channel attacks that may leak information, for instance using CPU caches [Per05, Aci07, AKS07]. Of course these side-channel attacks have also been studied in the context of virtualized and cloud infrastructures [RTSS09, ZJRR12]. The ability of co-locating the VMs of the attacker with the ones of the victim is a necessary requirement for a successful attack.

Software vulnerabilities contribute further to potential isolation failures. Exploiting a vulnerability in the hypervisor from a guest VM may lead to a privilege escalation and the failure of VM isolation. Such vulnerabilities have been studied in the literature [Orm07, PLS<sup>+</sup>14]. Practitioners of cloud deployments

---

often have security policies in place that restrict the co-location of different tenants to a specific subset of trusted hypervisors, or disallow co-location entirely.

---

### 1.1.2 Further Security and Operational Requirements

---

Isolation is an important goal, isolation failures a major risk, and one of the focusing security aspects of this dissertation. However, accidental or malicious configuration and topology changes may also result in other security and operational failures [OGP03].

The placement of virtual machines on particular physical servers is a requirement that is motivated by performance, legal, or also security reasons. For example due to data privacy reasons, VMs may only run on hosts of a certain geographic location [ENI09]. Creating new virtual machines on the wrong servers or migrating an existing VM to the wrong server may cause violations with regard to the placement policy. Of security concern is also the VM migration process itself. An attacker may manipulate data in transit if they have access to the network [OCJ08], or a VM is migrated to a host that is controlled by the attacker [RC11].

For VMs that host dependent services we want to make sure that these VMs can actually communicate with each other, e.g., they are connected on the network level and they are powered on. This is complementary to the isolation policy. Motivated by service dependability is also the absence of single point of failures, i.e., to offer sufficient redundancy [LDL<sup>+</sup>08]. For instance two replicated services are running on the same physical server [PZH13], which constitutes a single point of failure, or the network connectivity of the host is only realized through one network uplink. For even stronger failure resilience properties we want to make sure that replicated services are running on two different hypervisors, in order to minimize the impact of bugs in a particular hypervisor and to achieve independent faults. Similarly, services can be deployed on multiple independent clouds [Vuk10, BCQ<sup>+</sup>13] in order to achieve independent faults.

---

### 1.1.3 Problem Statement and Summary

---

Many different security challenges for both virtualized and cloud infrastructures are discussed in the existing literature. One important challenge are isolation failures in multi-tenant environments, which are caused by misconfigurations, malicious insiders, or system vulnerabilities. The complexity, heterogeneity, scalability and dynamics of virtualized infrastructures contribute to the root causes of isolation failures. In summary, the following problem has been identified:

**Problem 1.** *Complex, scalable, and dynamic virtualized infrastructures suffer from misconfigurations, malicious insiders, and system vulnerabilities that ultimately lead to security and operational failures, such as isolation breaches.*

This problem has been addressed in “traditional” IT infrastructures – at best – through the means of change planning and assessments of the networking infrastructure configuration. However, these methods only partially address the problem in a virtualized environment. Change planning does not cope with the dynamic nature of the infrastructure. Assessment of the network covers only one part of the virtualized infrastructure stack, which is composed out of network, compute, and storage.

If this problem remains unsolved, it is likely that both public cloud providers as well as enterprises will see further security and operational failures in their virtualized infrastructures as they continue to expand and increase in complexity. In fact, Chen et al. [CPK10] state that “*Given the stakes, it strikes us as inevitable that security will become a significant cloud computing business differentiator.*”. Werner Vogels, CTO of Amazon, confirms that security will always be their highest priority and their number-one area of investments [MIT14a].

---

## 1.2 Thesis

---

In light of the problem statement, I propose the following hypothesis:

**Hypothesis 1.** *It is possible to model and analyze complex, scalable, and dynamic virtualized infrastructures with regard to user-defined security and operational policies in an automated way.*

I pursue the following research questions in order to validate the hypothesis:

- Q1 How to model heterogeneous virtualized infrastructures with their configuration and topology? How to populate such a model in an automated way?
- Q2 What is a suitable isolation and information flow model? How to determine isolation among tenants in the infrastructure?
- Q3 How to express operational and security requirements? What requirements need to be expressed? What kind of formal foundations are suitable that enable an automated analysis?
- Q4 How to verify that the infrastructure — given as a model — fulfills the security requirements? What are existing analysis tools? How suitable, expressive, and efficient are they?
- Q5 How to cope with the infrastructure's dynamic behavior? How can we keep the infrastructure model up to date? Can we efficiently analyze changes happening in the infrastructure with regard to their security impact?
- Q6 Is it possible to prevent misconfigurations in the first place? How can we model configuration and topology changes in a virtualized infrastructure? How can we analyze them?

---

## 1.3 High-Level Approach and Scope

---

The goal of this dissertation is to develop a practical analysis framework for virtualized infrastructures that allows operators to express security and operational policies as well as to automatically analyze the infrastructure with regard to these policies.

Our focus are virtualized infrastructures in the context of private enterprise deployments. The analysis of public cloud infrastructures would be possible through a trusted third party, but is not the focus of this dissertation. We want to provide tool support for operators to enable them in assessing and maintaining the security of their infrastructure. The virtualized infrastructure comprises virtualized compute, network, and storage resources. It is the fundamental layer of the cloud computing stack that includes Infrastructure as a Service, which provides further automation on top of the virtualized infrastructure, as well as Platform and Software as a Service. It is crucial to ensure the security compliance of the fundamental layer, in order to provide a secure platform for the higher layers.

We consider both accidental as well as malicious configuration changes, thereby covering misconfigurations as well as insider threats. We do not differentiate between human operators performing configuration changes and agents, such as automation software, operating on the infrastructure.

Our analysis operates on a model of the infrastructure. The system capture both the current configuration and topology of the infrastructure in a model, as well as the operations that can change the infrastructure. This enables our analysis to assess the current and future states of the infrastructure. We cannot prove the correctness of the model itself, but we establish a methodology for the model population. The system extracts all the available configuration and topology data. For the data we explicitly decide which elements to translate into the model and which elements to ignore. All elements that are not explicitly processed result in a warning, which allow us to detect gaps in our model translation. For our operations model we use the API documentation of operations as well as empirical validation to study the effect of operations.

---

Our focus and scope is on the configuration and topology of compute, network, and storage resources. We are excluding the modeling and analysis of access control configurations and policies. Further we exclude software vulnerability analysis for both the hypervisor and virtual machines. The analysis operates on both the current, although dynamic, state of the virtualized infrastructure as well as possible future states. As future work we outline the analysis over sequences of states with temporal policies. We further treat virtual machines as black boxes, i.e., we perform no introspection nor analysis on what is stored or processed inside virtual machines. However, for many policies we require the operators to label and assign the VMs to security zones.

We perform a *static* analysis, i.e., we analyze a virtualized infrastructure based on its configuration and topology. For example, based on its configuration we compute potential information flows to determine isolation failures. However, we do not monitor at runtime the infrastructure for actual flows. Therefore our analysis bears similarity with static program analysis [NNH99], where the source code of a program is statically analyzed rather than the executed program. However we note that our analysis is also dynamic in the sense that when the infrastructure changes we also update our model and analysis. Our analysis must cover both the current state of the infrastructure as well as actual and intended changes. This means our analysis is event-driven and also tries to prevent misconfigurations in the first place. For the analysis we employ both custom tools as well as existing tools from the formal methods community, such as theorem provers and model checkers.

Our approach should not require modifications to existing virtualized infrastructures. Approaches such as information flow control systems often require modifications to the operating system. We aim for a simple deployment of our analysis framework in existing infrastructures. Furthermore, we aim for a framework that is almost fully automated. This is necessary to cope with such scalable and dynamic environments where we want to reduce the required involvement of human operators.

We evaluate the results of this approach through case studies in customer environments, with a laboratory semi-production infrastructure, as well as with simulated environments for scalability evaluations.

---

## 1.4 Contributions

---

In this dissertation we establish a new practical and automated security analysis framework for virtualized infrastructures. We propose a novel tool that automatically extracts the configuration of heterogeneous environments, builds up a unified graph model, and maintains this model upon changes in the dynamic environment. We are the first that lift static information flow analysis to the entire virtualized infrastructure, in order to detect isolation failures between tenants on all resources. Our analysis is configurable, using customized rules to reflect the different trust assumptions of users, as well as dynamic to adjust the information flow when the infrastructure changes. We propose a new generic analysis platform that allows to express a diverse set of security and operational policies in a formal language. We are the first to employ a variety of theorem provers and model checkers to verify the state of the virtualized infrastructure against the policies as well as analyze dynamic behavior of the system, such as VM migrations. Finally, we present a system for the run-time analysis of operations that can proactively mitigate misconfigurations using an operations transition model.

### Information Flow Analysis in Virtualized Infrastructures

We study the automated information flow analysis of scalable, heterogeneous, virtualized infrastructures. We propose a novel tool that is capable of discovering and unifying the actual configuration of different virtualization systems (Xen, VMware, KVM, and IBM's PowerVM). Our approach transforms the discovered configuration input into a graph model representing all resources, such as virtual machines, hypervisors, physical machines, storage and network resources. Further, the tool performs a static information flow analysis based on explicitly specified trust rules. We aim at reducing the analysis complexity for human administrators to the specification of such a few well-designed trust assumptions and leave the extrapolation of these assumptions and analysis of information flow behavior to the tools.

---

In summary, our analysis tool models virtualized infrastructures faithfully, independent of their vendor, and is efficient in terms of absence of false negatives as well as adjustable false positive rates. The analysis takes a set of graph traversal rules as additional input, which models the information flow and trust assumptions on resource types and auxiliary predicates. It checks for information flow by computing a transitive closure on an information flow graph coloring with the traversal rules as policy. From that, the tool diagnoses isolation breaches and provides refinement for a root causes analysis.

We evaluate our approach in a case study with a financial institution, which demonstrates the reasonable performance of our system as well as its ability to detect realistic isolation breaches in the infrastructure.

- Sören Bleikertz, Thomas Groß, Matthias Schunter, and Konrad Eriksson. Automated Information Flow Analysis of Virtualized Infrastructures. In *16th European Symposium on Research in Computer Security (ESORICS'11)*. Springer, Sep 2011.

### Virtualization Assurance Language

We contribute the first formal security assurance language for virtualized infrastructure topologies. More precisely, we model such an assurance language in the tool-independent Intermediate Format (IF) [AVI03], which is well suited for automated reasoning. We lay the language's formal foundations in a set-rewriting approach, commonly used in automated analysis of security protocols, with an extension to graph analysis functions. As a language aiming at expressing topology-level requirements, it can express management and security requirements as promoted by [DDLS01]. Management requirements in the cloud context are, for instance, provisioning and de-provisioning of machines or establishing dependencies. Security requirements are, for instance, sufficient redundancy or isolation of tenants. To test the expressiveness of our proposal, we model typical high-level security goals for virtualized infrastructures. We study the areas deployment correctness, failure resilience, and isolation, and propose exemplary definitions for respective security requirements.

- Sören Bleikertz and Thomas Groß. A Virtualization Assurance Language for Isolation and Deployment. In *IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY'11)*. IEEE, Jun 2011.

### Automated Verification of Security Policies

We are the first to apply general-purpose model-checking for the analysis of general security properties of virtualized infrastructures. We propose the first analysis system that can check the actual state of arbitrary heterogeneous infrastructure clouds against abstract security goals specified in a formal language. Our approach covers static analysis as well as dynamic analysis and employs a versatile portfolio of problem solver back-ends. We believe that our experiments with different analysis strategies (Horn clauses, transition rules) are of independent interest, because the problem instances for security assurance of virtualized infrastructures are structured differently than traditional application domains of model checkers, notably security protocols. In addition, we present early insights on the complexity relations of different problem classes.

As a case study, we successfully model checked a sizable production infrastructure of a global financial institution against the zone isolation goal. We have previously analyzed this infrastructure extensively with specialized tools and found the same problems with this generic approach. We report that our different optimizations allowed us to improve the performance by several orders of magnitude: whereas the non-optimized problem instances did not terminate within several hours, the optimized problem instances completed the analysis in the order of seconds.

- Sören Bleikertz, Thomas Groß, and Sebastian Mödersheim. Automated Verification of Virtualized Infrastructures. In *ACM Cloud Computing Security Workshop (CCSW'11)*. ACM, Oct 2011.

---

## Dynamic Information Flow Graphs

We propose the novel concept of information flow graphs constructed from user-defined flow rules. The flow rules capture trust assumptions on isolation in system components based on their attributes and connectivity. This leads to a generic and user-configurable approach that we apply to the case study of isolation in virtualized infrastructures. We analyze the correctness and complexity of our approach, in particular we adapt a firewall fault model to analyze flow rules sets.

We establish dynamic information flow graphs that are updated based on system model changes, including incremental, decremental, node property, and resulting connectivity changes. This enables a differential information flow analysis for dynamic systems. We apply our dynamic approach also to the case study of isolation in virtualized infrastructures in combination with a system that provides system model changes.

- Sören Bleikertz, Thomas Groß, and Sebastian Mödersheim. Dynamic Information Flow Graphs with Flow Rules. Technical Report RZ3893, IBM Research, 2016.

## Dynamic Infrastructure Model and Reactive Analysis

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

In order to establish such a differential security analysis, we propose an architecture that caters for near to real-time detection of configuration changes in heterogeneous virtualized infrastructures. We maintain a synchronized graph model of these infrastructures using a set of algorithms for the computation of graph deltas (added/removed nodes and edges, changed node attributes) applicable to a graph model based on change events. We offer a practical implementation of our system for VMware environments. Our evaluation shows that the differential approach reduces the overall analysis time significantly, putting near-to-real-time analysis in our reach. For a broad spectrum of cloud operations and even for large infrastructures, we measure model update times in the order of milliseconds, which renders our approach several orders of magnitude more efficient than previous static analysis approaches.

- Sören Bleikertz, Thomas Groß, and Carsten Vogel. Cloud Radar: Near Real-Time Detection of Security Failures in Dynamic Virtualized Infrastructures. In *Annual Computer Security Applications Conference (ACSAC 2014)*. ACM, Dec 2014.

## Operations Model and Proactive Analysis

We tackle the problem of misconfigurations and insider threats by establishing a practical security system that proactively analyzes changes induced by management operations with respect to security policies. We achieve this by contributing the first formal model of cloud management operations that captures their impact on the infrastructure in the form of graph transformations. Our approach combines such a model of operations with an information flow analysis suited for isolation as well as a policy verifier for a variety of security and operational policies. Our system provides a run-time enforcement of infrastructure security policies, as well as a what-if analysis for change planning.

We propose the first formal model of cloud management operations, the *operations transition model*, that captures how such operations change the infrastructure's topology and configuration. We express the operations as transformations of a graph model of the infrastructure, which is based upon the formalism of graph transformation [Roz97]. Further, we propose a unified model that integrates with the operations model the specification of security policies as well as an information flow analysis suited for isolation policies. We formalize a variety of policies, such as in the areas of isolation, dependability, and operational

---

correctness using graph matching. Finally, based on our model, we design and implement a practical security system which assesses and proactively mitigates misconfigurations and security failures in VMware infrastructures. We evaluate the performance of our analysis in a practical environment, and we further discuss and test the security of our system.

- Sören Bleikertz, Thomas Groß, Sebastian Mödersheim, and Carsten Vogel. Proactive Security Analysis of Changes in Virtualized Infrastructures. In *Annual Computer Security Applications Conference (ACSAC 2015)*. ACM, Dec 2015.

### Industry Impact and Other Publications

The research in this dissertation lead to the creation of a new product called *IBM PowerSC Trusted Surveyor* [BCD<sup>+</sup>13], which provides inventory and analysis of network isolation in IBM PowerVM-based virtualized infrastructures. Our contributions to the product line were awarded an IBM Research Division award. Furthermore, we published the following papers in the area of cloud computing security that are not part of this thesis:

- Sören Bleikertz, Anil Kurmus, Zoltan A. Nagy, and Matthias Schunter. Secure Cloud Maintenance - Protecting workloads against insider attacks. In *7th ACM Symposium on Information, Computer and Communications Security (ASIACCS'12)*. ACM, May 2012.
- Sören Bleikertz, Sven Bugiel, Hugo Ideler, Stefan Nürnberger, and Ahmad-Reza Sadeghi. Client-controlled Cryptography-as-a-Service in the Cloud. In *International Conference on Applied Cryptography and Network Security (ACNS 2013)*. Springer, Jun 2013.
- Sören Bleikertz, Toni Mastelić, Sebastian Pape, Wolter Pieters, and Trajce Dimkov. Defining the Cloud Battlefield - Supporting Security Assessments by Cloud Customers. In *IEEE International Conference on Cloud Engineering (IC2E 2013)*. IEEE, Mar 2013.

---

## 1.5 Outline and Organization

---

The remainder of this dissertation is organized as follows. In *Chapter 2* we explain the concept of virtualization and virtualized infrastructures. We review the existing literature in the areas of infrastructure discovery and modeling, policy languages for operational and security requirements, isolation and information flow analysis in particular in virtualization, as well as infrastructure state and change-based security analysis.

We first study the information flow analysis and policy-based verification of *static* virtualized infrastructure topologies and configurations. In *Chapter 3* we introduce a system that automatically extracts the configuration from heterogeneous virtualized infrastructures and performs a graph-based static information flow analysis to detect isolation violations. The approach is generalized to a range of user-defined policies by establishing a policy language in *Chapter 4* and the usage of model-checkers and theorem provers in *Chapter 5* to verify such policies against a given static infrastructure.

Building up on the analysis in the static case, we then study the analysis of *dynamic* virtualized infrastructures. In *Chapter 6* we introduce an information flow analysis using dynamic information flow graphs, which are computed and updated for a dynamic infrastructure topology. *Chapter 7* describes a system that analyzes configuration changes with regard to security policies in a *reactive* approach, i.e., after a change has been performed. In *Chapter 8* we introduce a system that operates in a *proactive* way, where changes are analyzed before they are deployed.

Finally, *Chapter 9* summarizes the contributions of this dissertation. We conclude by discussing the limitations of the proposed approach and outline directions of future work.



---

## 2 Background and Literature Review

---

### 2.1 Background

---

We will introduce in this section the concepts of virtualization, virtualized infrastructures, and cloud computing, and how these concepts are implemented in practical deployments.

---

#### 2.1.1 Virtualization

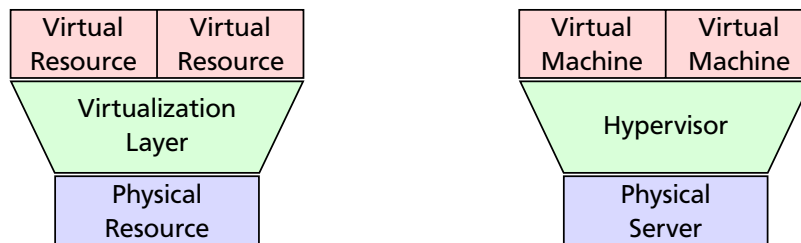
---

Virtualization is a way of multiplexing multiple virtual resources on top of a physical resource while preserving logical resource isolation, i.e., compartmentalization. Grandison et al. [GMTA10] define virtualization as the following:

*“A method, process or system for providing services to multiple, independent logical entities that are abstractions of physical resources, such as storage, networking and computer cycles.”*

Virtualization provides benefits such as rapid provisioning due to software-based resource orchestration, higher physical resource utilization, as well as flexibility by decoupling from the physical hardware. These are desired properties in modern IT infrastructures.

In the following we will explain in more detail different methods of virtualization for compute, storage, and network resources, which are widely deployed and commonly used. We will further discuss the composition of virtualization of different resources in larger infrastructures and the management of such virtualized infrastructures.



**Figure 2.1.:** Virtualization layers in its generic form and specifically for compute resources as a hypervisor.

---

#### 2.1.1.1 Compute Virtualization

---

Virtualization of compute resources date back to IBM mainframes in the 1960s and is now extensively used in modern IT infrastructures. The layer that provides compute virtualization is called a *hypervisor* or *Virtual Machine Monitor* (VMM). The hypervisor allows to run multiple *virtual* machines (VM) on a single physical server by sharing the physical compute and memory resources. The hypervisor preserves isolation between the virtual machines, although side- and covert-channels cannot be entirely prevented in a shared physical resources environment. The hypervisor is responsible for the entire lifecycle of the VM, including its setup and creation, termination, as well as migration to another physical server. Network and storage resources are also setup by the hypervisor for the VMs and thereby the hypervisor coordinates the entire provisioning of a VM.

We differentiate between three different kinds of compute virtualization: para, full, and hardware-assisted virtualization.

---

**Para-Virtualization** requires that the operating system in the virtual machine is aware that it is running in a virtualized environment. The guest system has been modified to communicate with the hypervisor, in particular for privileged operations that are not allowed to be executed by virtual machines directly due to security reasons. Proprietary operating systems have been lacking support for para-virtualization due to the required modifications. Xen [BDF<sup>+</sup>03] is one of the most common hypervisors that employs para-virtualization.

**Full-Virtualization** allows to run operating systems in VMs in an unmodified form, thereby also enables the virtualization of proprietary guest systems. The hypervisor has to translate the guest system instructions, again in particular for privileged operations. One benefit of such a instruction translation layer is that the architecture of the guest and the host systems may differ. QEMU [Bel05] is an example of a full-virtualization hypervisor that supports many architectures. Although instruction translation may result in negative performance impact for the guest, the hypervisor can also choose to directly execute safe instructions on the CPU, as realized in full-virtualized VMware ESX/ESXi and QEMU with the KQEMU [Bel08] acceleration extension. To provide I/O to guests, the hypervisor has to emulate common devices, such as network interface cards or storage controllers, that the guest system drivers support.

**Hardware-Assisted Virtualization** combines the performance of para-virtualization with the ability to run unmodified systems of full-virtualization. However, it requires the support of the CPU to provide compute virtualization, such as AMD-V and Intel VT-x, in particular to trap privilege operations in VMs. For guest I/O, the hypervisor can either provide device emulation, as in the case of full-virtualization, or the guests use para-virtualized drivers, such as based on virtio, which offer better performance. New generations of CPUs also provide an I/O MMU that allows to grant direct access of a VM to a hardware device, while restricting the use of DMA for isolation reasons.

With the increased availability of CPUs that offer hardware capabilities, hardware-assisted virtualization has become the dominating form of compute virtualization. In particular the recent performance improvements in the CPU's virtualization capabilities and the use of para-virtualized device drivers have made this form of compute virtualization the most efficient one.

Besides differentiating the method of compute virtualization, hypervisors are also classified into two types: type-1 or native hypervisors, and type-2 or hosted hypervisors. The first type runs natively on the hardware similar to a regular operating system kernel, whereas the latter type requires an existing operating system to run. In server virtualization, type-1 hypervisors are the dominant ones, whereas for desktop virtualization type-2 hypervisors are also common. In the following we discuss examples of widely deployed type-1 hypervisors:

**KVM** (Kernel-based Virtual Machine) [KKL<sup>+</sup>07] is an open-source extension to the Linux kernel that turns Linux into a type-1 hypervisor. QEMU can leverage KVM as an accelerator, and thereby replacing KQEMU, when the host and guest architectures are the same. QEMU is used to setup the VM and to provide device emulation, in case that virtio para-virtualized drivers are not used. Higher-level hypervisor and VM management is often done using libvirt. KVM with QEMU is the compute virtualization stack that is embraced by the Linux community.

**VMware ESXi/ESX** [VMw08] is a proprietary full- and hardware-assisted virtualization type-1 hypervisor. It provides an extensive API [VMw13a] to manage the hypervisor configuration and its inventory. Originally, ESX required a full Linux as a service operating system, but ESXi reduced that dependency and is now a small embedded hypervisor, with the goal of reducing the trusted computing base (TCB) and its attack surface.

**Xen** [BDF<sup>+</sup>03] is an open-source type-1 hypervisor that provides both para- and full-virtualization. The VM management and I/O is done in a privileged VM, *domain* in Xen's terminology, which is called *dom0*. A regular unprivileged VM is called *domU*. Dom0 is responsible for setting up and manage the lifecycle

---

of a VM by interacting through privileged instructions with the hypervisor. Furthermore, guest I/O is realized through dom0, where network bridging or routing is set up, as well as storage provided.

From a security point of view, the Xen hypervisor provides a relatively small TCB of hundreds of thousands source lines of code (SLOC). However dom0 is a privileged and trusted domain that often runs a fully-fledged Linux kernel and userland with millions of SLOC, which have to be considered in the TCB as well. Approaches to reduce the trust in dom0 have been proposed [BLCSG12, MMH08, BBI<sup>+</sup>13].

**PowerVM** [CCL<sup>+</sup>13] for PowerPC is an example for a non-x86 type-1 hypervisor. Unlike the previous example, this hypervisor is part of the firmware of the server. Inheriting from the mainframe virtualization, VMs are called logical partitions (LPARs). A privileged partition, the Virtual I/O Server (VIOS), provides network and storage I/O to the guests, similar to Xen's dom0.

In addition to the previous examples, type-2 hypervisors, such as VirtualBox and VMware workstation target desktop installations and not server virtualization in large-scale for enterprise infrastructures. Furthermore, operating-system-level virtualization with containers has seen increased popularity. The advantage compared to virtual machines is that containers require less overhead and the numbers of containers per physical host can be of one order of magnitude larger compared to VMs. Containers offer lower overhead but with weaker isolation, based for example on Linux cgroups. In order to provide adequate isolation, containers of one tenant are hosted within tenant VMs, in order to rely on the isolation provided by the VM.

---

### 2.1.1.2 Storage Virtualization

---

According to the general definition of virtualization, storage virtualization multiplexes virtual storage resources on physical storage, for example, hard disks. We differentiate between local and remote storage as well as file and block-based storage.

Local storage is only available to virtual machines that are running on the physical server that also contains the physical storage resources. In that case we have a convergence of compute and storage resources in a physical server. For remote storage, the virtual storage resources are provided over the network and attached by the hypervisor to the virtual machines. Local storage in public clouds is usually considered ephemeral, i.e., the content is only persistent as long as the VM is running on that particular physical machine. If the VM terminates, the associated local storage is also deleted.

Block-level storage offers a storage interface that operates on fixed-sized blocks. Physical storage, such as hard disks, provide such an interface with blocks, also called sectors, and block sizes of 512 or 4k bytes. Virtual disks that are attached to virtual machines operate on the same interface. On the other hand, file-level storage offers an additional abstraction layer on top of block-level storage and provides, among other features, data storage with varying lengths. In order to use a file as a backend for a virtual disk attached to a VM, we have to use a layer that provides a block-level interface on top of files, such as Linux's *loopback devices*.

#### **Local File-level Storage**

For local file-level storage we can simply leverage the existing file systems in the hypervisor. For each virtual disk of a VM a file is created to form the disks backend and we use a block-level layer on top of the created file, such as Linux's loopback devices. The advantages of this form of storage virtualization is that it is easy to deploy, since the existing file system can be used, and new virtual disks can be created by just creating new files. However, many abstraction layers, including the physical block device access, the file system, and the loopback device, negatively impact the virtual disk performance. Specialized file systems, such as VMware's VMFS, are optimized for virtual machine workloads and offer nearly direct block-device access performance.

---

## Local Block-level Storage

We can either map a physical block storage device to a VM or use a storage virtualization layer on top of the physical block devices that provide virtual ones. The first option can be realized using, for instance, VMware Raw Device Mapping (RDM) or VirtIO blk, and offers best performance but with limited flexibility. The other option is to virtualize the physical block storage devices into virtual ones. For example using the Logical Volume Manager (LVM) for Linux, we can aggregate multiple physical block devices into volume groups. From the volume groups we can flexibly provision logical volumes that form the backend of the virtual disks for VMs.

## Remote File-level Storage

Network-attached storage (NAS) is used to provide remote file-level storage based on established technologies such as NFS or CIFS. Files are used as virtual disk backends similar to local file-level storage, however in this case they are stored on the network file system and not locally. VMware's VMFS can also be used as a cluster file system that provides access to the same VM files for multiple hypervisors, but requires access to a shared network block-level storage, which is discussed in the next paragraph. New distributed network file systems have been proposed, such as CephFS [WBM<sup>+</sup>06], that provide better fault tolerance compared to existing technologies such as NFS or CIFS.

## Remote Block-level Storage

Storage area network (SAN) provides block-level access to a remote storage. Established technologies exist such as iSCSI and Fiber Channel (FC). The hypervisor discovers one or multiple of such network block devices and can either expose them directly to the virtual machines or use them in conjunction with a file system, such as VMFS. Systems based on established technologies, such as iSCSI or FC, often have limited scalability and fault tolerance. New systems that have vertical scalability, no single point of failures, and self-healing have been proposed such as Ceph RADOS [WLBM07] and Sheepdog [Mit14b]. VSAN [VMw15] provides similar properties for VMware environments.

---

### 2.1.1.3 Network Virtualization

---

Many methods exist to logically separate physical networks, in order to realize virtual networks. For example, distinct IP address spaces are assigned to realize a logical partitioning of the network. Firewalls and routers mediate and control traffic between elements of different IP address spaces. The network isolation and partitioning happens in *Layer 3* [Zim88], i.e., the network layer including IP. However, typically providers of multi-tenant environments want to offer *Layer 2* network partitioning, because the tenants can control their own IP configuration and addressing scheme.

A notable building block in network virtualization and *Layer 2* virtual networks are virtual switches, also called software network bridges. They offer the same functionality as physical network switches, but they are implemented in software and can be setup and configured through command-line tools or APIs. So called *Network Function Virtualization* [ETS12] applies the replacement of physical network elements with software not only to switches, but also to other network devices such as firewalls and load-balancers.

## Virtual Local Area Networks

A widely deployed way to realize logical *Layer 2* network separation is to use a Virtual Local Area Network (VLAN), which is defined in IEEE 802.1Q. A 32-bit field is inserted in the Ethernet frame and includes a 12-bit *VLAN identifier*, also called a *VLAN tag*. Logical separation of the physical network depends on the disjointness of the VLAN IDs. A VLAN ID of 0 indicates an untagged Ethernet packet and 4095 is reserved for implementation purposes, e.g., may represent a trunked port.

VLAN identifiers can either be configured and applied on the network switches or on the hosts. In traditional IT infrastructures, VLAN tagging has been realized on the switch level, because physical servers have been grouped into VLANs and the switches were centrally managed by the network staff, i.e., the

switches enforced the isolation. In a virtualized infrastructure, we are facing a more fine-grained grouping into VLANs: Multiple VMs on a single physical server may be grouped into different VLANs. In that case, the VLAN tagging must happen on the host level, in particular in the hypervisor, in order to group individual VMs into VLANs. Therefore, the hypervisor on the host-level applies the VLAN tags to the VM's network traffic.

In case of KVM and Xen, the Linux kernel is responsible for configuring the VM's network configuration and setting the VLAN tags. A virtual Ethernet device is configured that applies a given VLAN tag. The virtual device is used as an uplink for a software network bridge, to which a set of VMs are connected that belong to the same VLAN. Traffic between VMs on the same host will flow through the software network bridge. External traffic will flow through the uplink device where the VLAN tag is applied. In VMware ESXi, the virtual switches (*vswitches*) in the hypervisor [VMw07] provide virtual networking and also VLAN tagging. The *vswitches* are further divided into *port groups* that aggregate multiple connected VMs (virtual ports) under one common configuration. In particular, a port group configuration can contain a VLAN ID, which is applied to all the traffic of that particular port group, i.e., to the traffic of all VMs connected to this port group.

### Traffic Encapsulation

The number of virtual networks with VLANs is limited to 4094 (12-bit minus 2 for VLAN IDs 0 and 4095). In order to overcome this limitation, in particular in large multi-tenant cloud environments, and still providing Layer 2 networking to tenants, a new encapsulation approach called VXLAN [MDD<sup>+</sup>14] has been proposed. VXLAN encapsulates layer2 tenant traffic in UDP packets over IP and provides 24-bit VXLAN network identifiers to separate the virtual networks. Similar encapsulation mechanisms have been proposed. For instance, Generic Routing Encapsulation (GRE) [FLH<sup>+</sup>00] provides encapsulation of network layer traffic, e.g. IP, inside point to point IP tunnels. EtherIP [HH02] provides Ethernet frame encapsulation in IP tunnels. Despite the limitations of VLANs, they remain widely deployed, in particular in private virtualized infrastructures.

#### 2.1.1.4 Virtualized Infrastructure

A virtualized infrastructure is composed of the three building blocks of compute, network, and storage virtualization as well as an infrastructure management layer. Figure 2.2 illustrates a model of a virtualized infrastructure with a central management host and the different virtualization elements, for instance with VMs running on physical hosts and vswitches with port groups providing VLANs.

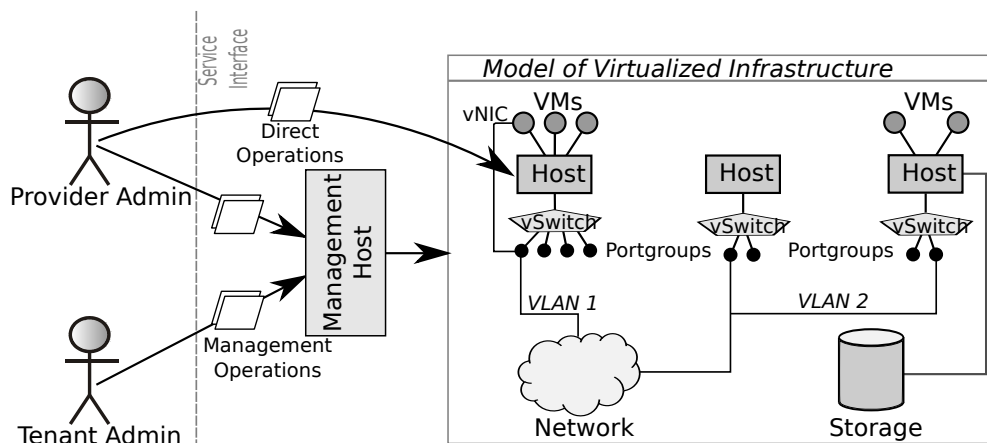


Figure 2.2.: Virtualized Infrastructure Model

---

## Infrastructure Management

So far we have presented the virtualization building blocks, but we want to be able to manage a large virtualized infrastructure, not just individual hypervisors and systems. Therefore, the management and resource orchestration is an important aspect.

We differentiate between *centralized* and *decentralized* management. In the centralized approach, management operations are performed through a central management host that manages the entire virtualized infrastructure. The management host is aware of all the resources and can orchestrate them according to the user requests. We have illustrated this architecture in Figure 2.2, where provider and tenant administrators send their management operations to the management host, which operates on the entire virtualized infrastructure. In a decentralized approach, the management operations are directly performed on the hypervisors and individual systems, which we illustrated in Figure 2.2 as *direct operations*.

Typical examples for centralized management is VMware with *vCenter*, which is a centralized management server that provides a web-service API. Administrators connect either with a proprietary client or programmatically to the API, in order to perform their operations on the infrastructure. In PowerVM environments we have a similar architecture with a *Hardware Management Console* (HMC) that provides a web-interface and SSH console for administrators to manage their PowerVM environment. On the other hand, Xen and KVM with LibVirt target a decentralized management, where individual hypervisors are managed. Aggregated management tools are built on top of this decentralized management, for example Xen's *XenCenter*, but these do not provide a central management host.

## Virtualized and Virtual Infrastructures

A *virtualized* infrastructure is a computing infrastructure that uses virtualization on physical resources, such as compute, storage, and network, in order to provision and provide virtual instances of these resources. A *virtual* infrastructure is composed of these virtual compute, network, and storage resources, which are provided through virtualization by the underlying infrastructure. Users are often unaware of the underlying virtualized infrastructures and are only concerned with their virtual infrastructure and the management of their virtual servers. Many virtual infrastructures for multiple tenants may exist on the same physical resources. The virtualized infrastructure encompasses the physical resources, the virtualization layers and their configuration as well as all the virtual infrastructures that are hosted.

---

### 2.1.2 Cloud Computing

---

Building upon the concepts of virtualization and virtualized infrastructures, Cloud Computing has become a widely used model of service and IT resource delivery. NIST [MG09b] defines *Cloud Computing* as the following:

*“Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.”*

Virtualization is a key technology that enables such sharing of resources, rapid provisioning, as well as compartmentalization of different tenants. Cloud computing and in particular cloud management software provides fully automated resource orchestration on top of a virtualized infrastructure. NIST defines the following cloud deployment and service models [MG09b]:

#### Deployment Models

NIST introduces multiple deployment models and distinguishes between private, community, public, and hybrid clouds.

In private clouds, the infrastructure is exclusively used by a single organization, although the organization can be composed of multiple business units, which are cloud tenants on their own and should be isolated

---

from each other. The cloud infrastructure must not necessarily be owned and managed by the organization itself, but can be operated by a third-party. A community cloud is used by a community of organizations that share common requirements, such as security or compliance related.

Public clouds offer cloud infrastructure to the general public. The infrastructure is operated and typically also owned by the cloud provider. Multiple tenants, potentially also conflicting ones, share the same infrastructure and are only logically isolated from each other. Some public cloud providers also offer *virtual private clouds* that provide stronger network isolation and also the possibility of exclusively using physical servers without the co-location of other tenants. Virtual private clouds are often integrated into the cloud consumers on-premise infrastructure through a VPN, thereby forming a hybrid cloud infrastructure.

### Service Models

There exists multiple abstraction levels to provide a service. The common ones are Infrastructure (IaaS), Platform (PaaS), and Software as a Service (SaaS), although other service types have been proposed as well, which can be described as *XaaS*: everything as a Service.

IaaS, or also called Infrastructure Clouds, offer the provisioning of compute, network, and storage resources. The cloud consumer has control over the computing environment in terms of what operating system and applications are installed on which virtual servers, as well as the virtual servers network and storage resources. The consumer however has no control over the underlying virtualized infrastructure in terms of controlling on which physical servers their VMs are provisioned. However in private cloud infrastructures, the provider and consumer come from a single organization and may cooperate to provide the consumers more insights on the underlying infrastructure in case of security and compliance requirements. A common open-source cloud management software for private infrastructure clouds is OpenStack.

PaaS supports developers to build and run web services. The cloud consumer deploys applications using programming environments, libraries, services, and other tools provided by the cloud provider. Common application infrastructure like databases are provided in a scalable and managed way to the application developers. In PaaS, the cloud consumer does not control the underlying computing infrastructure (servers, networks, storage), but only has control over the application environment. Examples for PaaS include on the open source side Cloud Foundry and on the commercial side Heroku, Google App Engine, and IBM BlueMix.

In SaaS the cloud consumer uses a provider's application, which is running on a cloud infrastructure. The application is accessible over the network and typically consumed with a web browser. Moving higher in the abstraction levels, in SaaS the cloud consumer neither control the infrastructure as in PaaS, nor the application environment (operating system, individual application capabilities). Typical examples of SaaS are Google GMail and Salesforce CRM.

The different service levels often build on top of each other. For example, a startup that offers a new application in a SaaS service model may use Heroku to deploy their application, and Heroku deploys their PaaS stack on top of Amazon Web Services cloud infrastructure (IaaS). This renders Infrastructure as a Service as the critical and fundamental basis on which many other applications and platforms depend on. Security problems in the cloud infrastructure layer, or in the virtualized infrastructure that is an integral part of this layer, can negatively impact the security of many tenants as well as their applications and users.

---

## 2.2 Literature Review

---

We review the literature and existing work in the areas of i) infrastructure topology and configuration discovery, monitoring, and modeling, ii) policy languages for operational (configuration) and security aspects, iii) isolation and information flow analysis, as well as iv) infrastructure state and change security analysis and planning.



Figure 2.3.: Overview of Literature Review.

## 2.2.1 Configuration Discovery and Modeling of Networks and Systems

### Non-Virtualized Infrastructure Discovery

The infrastructure discovery in non-virtualized environments has produced a variety of systems that cover different aspects of the infrastructure including physical network topology [BGJ<sup>+</sup>04, BBGR03, LOG01], routing topology [SMWA04, HPMC02], dependency of applications [CZMB08, CKG<sup>+</sup>08, KGE06], and application storage [JPRD10]. For a survey on further discovery methods we refer to [DF07].

### Graph-based Modeling of Networks

The modeling of network topologies is often based on graphs as they intuitively capture the relational aspects of the network [CDZ97]. These models can use graphs in different forms, e.g., they can be directed or undirected, edge-attributed to represent network bandwidth or costs, in order to capture the specific problem domain. Rules of routers and firewalls are as well modeled as forwarding graphs [KZZ<sup>+</sup>13] or as binary decision diagrams [ASMEAE09], which are an encoding of a Boolean function as a directed acyclic graph.

### Virtualized and Cloud Infrastructures

A tendency in virtualized data center is that the complexity in the physical network shifts into the virtual one. The physical network configuration is simplified and its topology flattened, whereas complex



---

virtual networks are created. Approaches for the discovery of the virtual network topology have been proposed [BHK12, NMCS10]. In public infrastructure clouds, the configuration of the firewall setup has been extracted and modeled as a graph [BSP<sup>+</sup>10]. The placement of VMs has been analyzed in public infrastructure clouds through network probing and co-location verification [RTSS09, HSUW13]. Inspecting the configuration inside a virtual machine can be performed during runtime using virtual machine introspection [PdCL07, MKLT13] or statically by analyzing the VM images [WZA<sup>+</sup>09]. An application and service centric graph model has been proposed for cloud infrastructures [BFL<sup>+</sup>12] with an automated discovery system [BBKL13].

### **Monitoring of Dynamic Networks**

IT infrastructures tend to become more and more dynamic. Existing work on the discovery and monitoring of dynamic environments has focused on the network level, for instance on routing [AFBB02, KVCP97] or peer-to-peer overlay networks [RIF02].

### **Dynamic Virtual and Cloud Infrastructures**

Virtualization lead to this dynamic behavior also for compute and storage resources. We need to be able to monitor and model changes for network, compute, and storage resources in virtualized infrastructures. On the hypervisor level, Cloud Verifier [SSVJ13] detects changes in the integrity of the hosted VMs on behalf of cloud customers. The system can also be used by providers to monitor the integrity of their cloud platform. From a performance point of view, vQuery [SGG12] is a system for monitoring VMware environments that stores partial topological information in a graph model and tracks many performance metrics. In public infrastructure clouds, Amazon recently introduced a service, called AWS Config [Ama14a], that provides change events to customers for their virtual environments. However, it provides no insights on changes in the underlying infrastructure, e.g., on VM migration and placement.

### **Summary**

Many discovery approaches for physical infrastructures (network topology, routing etc.) have been proposed. It is necessary to complement the existing view on the physical infrastructure with a view on the virtualized one. The challenge is to support heterogeneous virtualized environments that cover the entire stack of compute, network, and storage resources, as well as to cope with its dynamic behavior and frequent changes. Since graph-based models are widely used for capturing network topologies, they are possibly suited for virtualized infrastructure topologies as well.

---

## **2.2.2 Isolation and Information Flow Analysis**

---

Isolation is important in multi-tenant virtualized infrastructures. We review existing work in the areas of isolation and information flow analysis and control.

### **(Decentralized) Information Flow Control**

Information flow control (IFC) [Den76] is a mandatory access control approach that governs how information may flow between different storage objects based on their security classes. Decentralized IFC (DIFC) [ML97] introduces security labels for each object that, for instance, denotes its allowed readers. Thereby following an approach close to discretionary access control. DIFC has been implemented in new operating systems such as HiStar [ZBWKM06] and Flume [KYB<sup>+</sup>07].

### **Access Control Security Models**

Multiple access control models have been proposed that govern how information may flow between storage objects. These include Bell-LaPadula [BLP76] for confidentiality, Biba [Bib77] for integrity, type enforcement [BK85], Clark-Wilson integrity [CW87], and Chinese wall [BN89].

---

## Reachability Analysis in Networks

Isolation analysis in networks is based on network reachability and firewall filtering, which has been extensively studied in [ASMEAE08, KL09, KSS<sup>+</sup>09, XZM<sup>+</sup>04] and [MK05, MWZ00, Woo01] respectively. Reachability analysis in dynamic networks has also been demonstrated [KZZ<sup>+</sup>13] as well as algorithms that enable efficient reachability queries in dynamic graphs [RZ04].

## Hypervisor Isolation and Security

In virtualized environments, a major focus is the study of isolation on the hypervisor level. Kelem and Feiertag apply the Separation Model [Rus82] to model a secure virtual machine monitor [KF91]. In practice such strict isolation between VMs is not desired, but a controlled and mediated communication. sHype [SJV<sup>+</sup>05] is an extension to the Xen hypervisor that provides MAC policies such as Chinese wall to prevent side channel attacks and type enforcement for sharing resources. In practice, side channels between VMs, e.g., based on CPU caches [Aci07, Per05], has been investigated [RTSS09] and defense mechanisms for the hypervisor have been proposed [ZR13, ZJOR11].

Furthermore, many new hypervisors and security architectures have been proposed to protect applications and VMs against malware or malicious administrators [GPC<sup>+</sup>03, ZCCZ11, MLQ<sup>+</sup>10]. A detailed comparison of a subset of these secure hypervisors is given by Vogl [Vog]. In general, introducing a new critical piece of system software, such as the hypervisor, also increases the potential of vulnerabilities which may lead to privilege escalation and isolation breaks [Orm07, Woj08]. Architectures with minimal [SK10] or no hypervisor [KSRL10], with self-protection [WWGJ12, WJ10], as well as formally verified hypervisors [KEH<sup>+</sup>09], have been proposed to reduce the attack surface and harden the hypervisor. Besides seL4 [KEH<sup>+</sup>09], further models of hypervisors have been proposed to study their security properties including isolation [BBCL11, FM11, HLC<sup>+</sup>13].

## IFC in Cloud and Virtualized Infrastructures

Moving from the hypervisor to higher levels of the virtualization and cloud computing stack. Bacon et al. provide an overview and discussion on the usage of information flow control in secure cloud computing [BEP<sup>+</sup>14]. They conclude that DIFC is particularly suited for Platform as a Service (PaaS), because they believe this layer is best suited to provide labeling information on data and that existing open source implementations can be augmented. In fact, many approaches for information flow control on the PaaS level have been proposed. SilverLine [MRF11] offers data and network isolation based on data labeling, however requires changes to both Xen and the guest VM kernel. SilverLining [KHK14] focuses on Java application in the context of Hadoop map-reduce jobs and offers IFC through aspect oriented programming. It uses an information flow graph that captures users and files as nodes with read/write or disallowed flow directed edges. CloudSafetyNet [PMOK<sup>+</sup>14] monitors information flow through HTTP tagging on the client side and socket monitoring. CloudFence [PKZ<sup>+</sup>13] offers data flow tracking on the byte level using binary instrumentation. They claim to be more fine-grained than previous approaches that operate on the process level, such as SilverLine. CloudFlow [BFB<sup>+</sup>14] is based on VM introspection to monitor and extract the tasks with their SELinux security labels that running inside a VM. They implement a Chinese wall policy to prevent, for instance, that a VM with unlabeled tasks is running on the same physical server as a VM that runs a top secret task.

IFC also finds application in infrastructure management and administration. For instance the Chinese wall policy is implemented for administrators that log into customer virtual machines, in order to prevent an administrator to log into VMs of conflicting customers [WAHS10]. Further, *H-one* [GL12] is a system that uses information flow tracking to establish an audit log of VM configuration tampering by administrators.

## Isolation Architectures in Virtualized Infrastructures

There exists also approaches that deploy the infrastructure in such a way that it provides tenant isolation. The Trusted Virtual Datacenter (TVDC) [BCP<sup>+</sup>08] offers automated deployment with isolation and integrity by leveraging a trustworthy hypervisor, trusted computing, and automated setup of network

---

isolation [CDRS07]. The system uses Trusted Virtual Domains (TVDs) [CDE<sup>+</sup>10] to group resources together and defines an information flow policy, which is a similar concept to IFC. A formal isolation model for TVDc is presented in [BKS14]. On the network level, CloudPolice [PYK<sup>+</sup>10] provides network access control in the hypervisor and essentially implements a distributed firewall. Network architectures that ensure tenant isolation while also integrating with the customers on-site network exists [HLMS10, WGR<sup>+</sup>09]. IFC approaches do not protect against side channel attacks, which recently have been demonstrated in PaaS [ZJRR14], and additional security approaches are required. In particular, the enforcing of cache isolation [RNSE09] is important as many side channels rely on the shared cache, as well as to provide a VM placement algorithm that offers co-location resistance [AKM<sup>+</sup>14]. Falzon and Bodden [FKBE15] propose a framework to provide isolation on multiple layers, such as on the virtual CPU, process and containers, and virtual machines. The framework migrates processes and virtual machines when a policy is violated, for instance, when unusual activity is detected by a probe.

### Summary

Mandatory access control approaches for information flow as well as policies have been extensively studied and applied, for instance, in the design of new operating systems. Similarly, reachability analysis is employed in networks to determine isolation of network components. In the context of virtualization, isolation has been studied on the hypervisor level as well as application level. We need to lift information flow analysis to the entire virtualized infrastructure level and be able to cope with the different trust assumptions, e.g., side channel resistance on particular secure hypervisors. Information flow control approaches often require modifications to the software components, however less invasive approaches, such as static analysis, have not been adequately studied in the context of virtualized infrastructure topologies and configurations.

---

## 2.2.3 Formal Languages for Security and Operational Policies

---

We review formal languages for both configuration and infrastructure management as well as languages for security policies.

### Configuration and Deployment

Policy-based infrastructure management is becoming more popular as infrastructure automation increases. On the operating system level, such policies describe for instance configuration files and installed packages. Tools such as Chef [che], Puppet [pup], and CFEngine [cfe, Bur95] offer either proprietary or embedded languages to describe the configuration state.

In particular for VM deployments, the *Open Virtualization Format* (OVF) [DMT10] is a standardized specification language for the packaging and distribution of virtual machines. OVF is used to describe general information and virtual resource usage for an individual virtual machine or a virtual appliance consisting of multiple VMs, but not for an entire virtualized infrastructure. PoDIM [DJ07] is a configuration language that not just focuses on individual operating systems or virtual machines, but one that covers high-level cross-hosts configuration management. Similarly, TOSCA [OAS13] is a standardized XML-based language to describe service topologies in a cloud computing environment.

Configuration management and languages for network devices has been subject to extensive research and many systems have been developed [CMVdM09, EMS<sup>+</sup>07, BF07]. The recent advancement in software-defined networks (SDNs) has also lead to the creation of many new configuration and flow specification language, such as Procera [VKF12], Pyretic [MRF<sup>+</sup>13], FML [HGC<sup>+</sup>09], which are also event-driven and can change the configuration upon topology changes.

### Access Control Policies

Many access control policy languages exist and we only discuss a small subset of languages which are related to virtualized infrastructures, such as distributed and dynamic systems.

---

*Ponder* [DDLS01] is an object-oriented formal specification language for access control policies and role management in distributed systems. However, it does not aim at expressing high-level security goals for infrastructure topologies and configurations. A graph-based approach for security policy specification, in particular for access control, has been presented in [HPL98] that expresses the policy as a directed graph with annotations. A policy graph contains a domain, which matches part of the system also given as a system graph, and a requirement that indicates restrictions on the system by the policy.

Kagal et al. [KFJ03] present a policy language for pervasive computing, which is similar to cloud computing environments with regard to their dynamic behavior. The language is motivated by access control and is used to express entitlements on actions, services, or conversations of an entity, such as an agent or user. Their implementation is based on Prolog. Further languages include XACML [OAS05], a XML-based language for attributed and role-based access control policies, SPL [RZFG99], an event-based access control language, as well as Datalog-based languages [DeT02, BFG10]. Fable [SCH08] is a security labeling and enforcement language in the context of language-based security. They formalize access control and static information flow policies (non-interference) as examples in Fable. Similarly, Fine [SCC10] is a language for access control and information flow policy specification in programs with an automated analysis. Binder [DeT02] is an extension of Datalog to specify access control in distributed systems. We refer to [HL12, DBSL02] for further comparison and discussion on policy languages for access control and management.

### Virtualization and Infrastructure Security

The security specification of VMs can be achieved with the concept of *Virtual Machine Contracts* [MGHW09], which are a policy specifications based on OVF that govern the security requirements of a virtual machine, e.g., to specify firewall rules. Similar to OVF, the objective of this language is linked to provisioning rather than expressing high-level security goals on the topological level. On the hypervisor level, *sHype* [SJV<sup>+</sup>05] is an implementation of access and isolation control for virtual machines, which uses a XML-based access control policy<sup>1</sup>. Again, the policy only applies to one entity in the virtualized system, i.e., the hypervisor hosting virtual machines. Xenon [MML<sup>+</sup>12] provides XML-based policies for inter-VM communication, MAC with VM labels to implement Chinese wall, type enforcement, and time-based rules, as well as policies on hypercall and resource usage.

On the network router and firewall level, Bartal et al. propose a model definition language [BMNW04] to describe security zones, network topology, and firewalling. Hinrichs proposed a policy framework [Hin99] that expresses policies as conditions leading to actions. The conditions can test on packet headers as well as global conditions, such as time and network load, which however require access to such live-data through the policy framework. Furthermore, the conditions can also operate on extended state associated with network flows, e.g., the association of users with source IPs. Actions include filtering (permit, deny), cryptographic requirements (encrypt traffic), and QoS. CloudPolice [PYK<sup>+</sup>10] is a distributed firewall with policies on tenant isolation, inter-tenant communication, and QoS (fair sharing, rate limiting). The policy format resembles the policies of Hinrichs [Hin99] where conditions lead to actions. A condition is a logical expression of predicates on properties such as sender/receiver, packet header fields, time, and past traffic state. And action can allow, block, and rate limit matched flows.

Motivated by security in SDNs, the Flow-based Security Language [HGC<sup>+</sup>08] allows the specification of high-level allowed network flows. *Alloy* [Jac02] is a first-order logic modeling language, which is used, among other things, in network infrastructure modeling and analysis [Nar05a, NCPT06]. Alloy can express structural properties as relations between objects as well as temporal aspects as dynamic models with states and transitions.

Data-centered policies have been proposed in the context of cloud computing and virtualized infrastructures. Santos et al. [SRGS12] proposes policy-based sealing of data, i.e., encrypted the data under an associated state and certain properties of the state (cf. [SS04]). The policy language is not fully defined,

---

<sup>1</sup> Xen User Manual, Section 10.3.

---

but the examples indicate logical expressions of equality and inequalities on state property key-value pairs. For example, the hypervisor must be equal to *CloudVisor* [ZCCZ11] and the version greater or equal to 1.

### Formal Verification

A large body of work exists on languages for formal verification of distributed systems, services, and cryptographic protocols, to only name a few use cases.

TLA+ [Lam02] is a formal language to specify distributed and concurrent systems. On a high-level, the language allows to specify an initial state of the system, state transitions, as well as invariants on states and temporal formulas. Amazon uses TLA+, and its variant PlusCal, to specify a variety of distributed system algorithms [NRZ<sup>+</sup>15, New14]. PROMELA [Hol91] is a language to express processes and their message passing in order to specify and verify parallel programs. The SPIN [Hol97] model checker consumes PROMELA specifications and can check for the absence of deadlocks, state invariants, and LTL constraints.

In the area of cryptographic protocols specification and verification, the language Intermediate Format (IF) [AVI03] allows to specify an initial state of a system as a set of facts, rewriting rules leading to state transitions, and goal rules that try to match against states. ASLan [AVA07] is an extension of IF and includes the specification of Horn clauses. These languages are supported by a wide range of analysis tools, such as, OFMC [BMV05a], SAT-MC [AC04], and other tools from the AVISPA project [ABB<sup>+</sup>05].

### Summary

Policy languages for security and operational aspects of virtualized infrastructures need to incorporate concepts from many different aspects such as configuration and deployment patterns, reachability and isolation specification, as well as state transitions and state invariants. The goal is not to develop another policy from scratch specifically for the context of virtualized infrastructures, but to build up and extend existing languages.

---

## 2.2.4 Security Analysis of Infrastructure States and Changes

---

This section is complementary to the analysis approaches that focus on isolation and information flow properties as previously discussed. In this section we review existing work in the areas of integrity verification in virtualized infrastructures, vulnerability analyses of networks and systems, as well as the security analysis and planning of configuration changes.

### Integrity Verification and Enforcement in Virtualized and Cloud Infrastructures

The integrity of the infrastructure in virtualized and cloud environments is fundamental for the security of the workloads and applications running on top. We review approaches for the verification and enforcement of integrity in such infrastructures. In particular, we look at the different layers ranging from operating system integrity, to VMs and hypervisors, up to the entire virtualized infrastructure. A fundamental mechanisms in many of such systems is the usage of trusted computing, in particular, the usage of remote attestation as offered by *Trusted Platform Modules* (TPMs).

On the operating system level, integrity is measured and can be verified based on remote attestation using TPMs. For instance, the *Integrity Measurement Architecture* (IMA) [SZJvD04] measures not only the boot-loader and kernel, but also the applications and its files. Building upon IMA, the PRIMA [JSS06] system enables integrity verification on the OS-level for information flow policies.

Terra [GPC<sup>+</sup>03] was one of the first virtualization platforms that incorporates TPM to provide integrity for VMs. In particular, a trusted hypervisor provides confidentiality and integrity to “black-box” VMs, and enables remote attestation using the TPM. However, it only provided load-time attestation. Overcoming the limitations of load-time integrity, the HIMA [ANSZ09] system provides run-time integrity monitoring for VMs using a hypervisor-based measurement agent.

---

A variety of approaches have been proposed that enable integrity monitoring and verification not only for VMs but also for the hypervisor. For instance, HyperSentry [ANW<sup>+</sup>10] uses the System Management Mode (SMM), which operates on a more privileged level than the hypervisor, to perform integrity measurements. The measurements are triggered through an out-of-band channel using IPMI. HyperSafe [WJ10] provides self-protection for the hypervisor and focuses on control-flow integrity (CFI). Finally, CloudVisor [ZCCZ11] uses nested virtualization to introduce a higher privileged layer under the hypervisor for the integrity protection.

In addition, MAC policies implemented by the hypervisor, such as sHype [SJV<sup>+</sup>05], have been analyzed with regard to integrity [RVJ09]. They model SELinux policies as an information flow graph and try to find violating paths.

Moving towards integrity verification and measurements of an entire virtualized infrastructure and not only individual hypervisors or VMs. The TCCP [SGR09] system consists of a trusted hypervisor and a trusted coordinator. Similar to Terra, the trusted hypervisor provides integrity and confidentiality for VMs. The trusted coordinator manages a set of trusted nodes based on their attestation results. The coordinator creates and migrates VMs only on the set of trusted nodes. Similarly, Krautheim [Kra09] proposes a security architecture with dedicated measurement VMs that uses virtual TPMs [BCG<sup>+</sup>06]. The CloudVerifier [SSVJ13] architecture enables end-users to verify cloud infrastructures. The system combines multiple existing components and approaches such as the Integrity Verification Proxy (IVP) [SVJ12], which performs remote attestation and VM integrity measurements, as well as trust anchors [SMV<sup>+</sup>10] that establishes transitive trust in the cloud infrastructure.

### **Network and System Vulnerability Analyses**

Ritchey et al. [RA00] employs model checking to analyze vulnerabilities in systems and networks. They model the following properties: 1) hosts with their vulnerabilities and current access levels from an attacker point of view; 2) available exploits with their conditions and resulting access level escalation; 3) a connectivity matrix for the hosts; 4) failure definitions stating invariants for the system in temporal logic. There exists potential scalability problems with their modeling approach, for instance, the number of exploits is fixed, they have to perform explicit connectivity checks in their exploits, and the connectivity matrix is pre-computed for all pairs of hosts.

MulVal [OGA05] uses Datalog for modeling and reasoning on multi-host and multi-staged vulnerabilities in networks. The system integrates with an automated vulnerability scanner to obtain vulnerability information of hosts and services. The scanner further provides information of the host, such as running network services, existing client and setuid programs, file ownership and network filesystem exports. In combination with a vulnerability database that describes the exploit range (local or remote) as well as the exploit's consequences, such as privilege escalation, the reasoning system can find violations of data access policies. The system can further perform what-if analyses on hypothetical vulnerabilities. In comparison to the work of Ritchey, MulVal uses a largely automated approach for collecting system configurations and vulnerabilities. However, it still misses the automated extraction of hosts connectivity, although they envision to rely on existing firewall and network analyzers.

Phillips and Swiler [PS98] propose a network vulnerability analysis based on attack graphs. The attack graphs are automatically generated from a set of attack templates, system configuration information, and attacker profiles. The attack templates describe individual attacks with their preconditions, state, and actions. The attacker profile captures the attacker capabilities in order to determine the probability of success of attacks. Finally the system configuration describe the physical network topology, although to automatically discovered, as well as the configuration of individual hosts. The attack graph contains weighted edges, where the weights captures success probabilities or cost of an attack, and a weighted path finding algorithm finds the low-cost attacks.

Ammann et al. [AWK02] provide a more efficient representation of attack trees or graphs. The nodes, or also called attributes, describe facts on vulnerabilities and attacker access. The edges describe exploits that require a set of attributes and produce a set of attributes. The attacker goals are a set of desired

---

attributes and the analysis algorithms create a directed exploit graph, find the minimal sequences of exploits, and obtain the required exploits to reach the goal attributes.

Tong et al. [TPT<sup>+</sup>10] study the vulnerability of general large graphs, such as computer networks, based on structural aspects of the graph. They compute a single vulnerability value based on the adjacency matrix of the graph, and define a shield value of a subset of nodes to capture the vulnerability resistance.

Madi et al. [MMW<sup>+</sup>16] model virtualized infrastructures as graphs capturing the physical, virtual, and cloud layers. Security properties, such as resource isolation and co-residency, are formalized in first-order logic and analyzed using a CSP solver.

AWS Trusted Advisor [Ama] provides automated checks for AWS infrastructures for a variety of operational and security properties. The security checks include for instance firewall and access control configurations. In addition to the Trusted Advisor's firewall configuration analysis, the analysis in [BSP<sup>+</sup>10] combines the firewall configuration with system vulnerabilities and constructs topological attack graphs. Their policies govern paths between two vertices in the graph with certain vulnerability rankings. Either paths should be entirely absent or paths up to a certain vulnerability ranking are tolerated.

### **Dynamic Virtualized and Cloud Infrastructures**

Misconfigurations in networks have been a problem in the operation of IT environments for a long time and solutions have been proposed. Mahajan et al. [MWA02] studied misconfigurations in BGP routing configuration changes by listening to changes and assess these. Kim et al. [KBAF11] analyzed the evolution of network configurations by mining a repository of network configuration files.

With the rise of software-defined networking, real-time monitoring and policy checking have been achieved in these environments [KCZ<sup>+</sup>13, KZZ<sup>+</sup>13]. CloudWatcher [SG12] is a system that enforces mediation of network flows by policy. The administrator defines a set of security network devices with their capabilities. The security policy, called the security language interface, defines a flow condition and a set of security devices, which need to be traversed by the flow. They propose four algorithms to find an optimal routing path for a given flow that routes the flow over the required security devices. Bellessa et al. [BKF<sup>+</sup>11] proposes a system called NetODESSA that performs dynamic policy enforcement in virtual networks. A dynamic network policy governs not individual hosts, but is a high-level policy that operates on hosts with specific characteristics. The system monitors new flows and registers a decision by the reasoning engine based on the policy with the OpenFlow switches.

Al-Haj and Al-Shaer [AHAS13] propose a framework for modeling and analysis of VM migrations. They model the migration as a constraint satisfaction problem, where for a given VM placement a migration sequence needs to be found that satisfies the desired VM placement and additional constraints. The constraints capture dependency, performance, and requirements, such as, the prevention of co-location of conflicting VMs. They use a SMT solver to find a sequence of migration steps that satisfy the constraints.

Cloud Calculus [JED<sup>+</sup>12a] is a formal framework for the modeling and analysis of VM deployments, in particular maintaining security invariants given by firewalls during VM migrations. The model is based on the mobile ambient calculus, a process calculus, that models VM mobility as well as firewall configuration changes. The preservation of policies during VM migration must ensure that packets are treated the same before and after migration. Follow-up works, which are extending Cloud Calculus, include the modeling and analysis of distributed firewalls as well as intrusion detection systems and VPNs. Jarraya et al. [JED<sup>+</sup>12b] extends the Cloud Calculus to cover distributed firewalls through composition of firewall configurations. Eghtesadi et al. [EJDP14] uses the Cloud Calculus approach to focus on the preservation of monitoring and tunneling configurations in IDS and VPN settings.

Focusing on operational problems as part of VM migration and configuration changes, CloudInsight [AJ11] tracks VM configuration changes and correlates changes with performance problems. They model and monitor physical as well as virtual machine configurations through a polling agent on the hypervisor. If a problem has been reported for a VM, the system identifies suspected change events based on the instability of a change. The suspected events are ranked based on their sensitivity, i.e., how often an attribute

---

changes, in a local (VM instance) and global view. The most likely root causes are then interactively remediated with the help of the user.

Generalizing the analysis and planning of VM migration to changes in general, we now discuss approaches for network and virtualization infrastructure change planning. Change planning and the analysis of changes for network routers has been addressed by existing research [TZV<sup>+</sup>08, AWY08], focusing in particular on performance properties.

Hagen [Hag13, HSK12] studies the verification of change operations in the context of statically and dynamically routed networks. They operate on objects and attributes of infrastructure components tracked in a CMDB. A change is modeled as a set of predicates and a set of effects on those attributes. A case study investigates the change of network routing from high-capacity to low-capacity network. They model change operations for taking down an interface and shifting traffic. A safety constraint specifies that to only route high-capacity traffic via high-capacity routers.

Kikuchi [Kik13, KH14] studies configuration synthesis and vulnerability analysis of dynamic virtualized and cloud infrastructures using formal methods. For the configuration synthesis, they model four operations: establish a physical connection, give access to another component, join a VLAN, and VM migration. The synthesis is further guided by constraints on physical and virtual network connectivity as well as host capacity. Given a goal condition, the Alloy Analyzer tries to find a sequence of operations that satisfy the constraints and reach the goal condition. The initial state of the system is translated from a configuration database (CMDB) into Alloy. With regard to operational vulnerability analysis, Kikuchi focuses on services availability, in particular due to single point of failures, over capacity, and load-balancer misconfigurations. They model changes to the infrastructure such as crash faults, VM migrations, and monitor changes of a high-availability load-balancer. As part of the state transitions they model the dependency among the components, for instance, if a physical server fails then all the VMs running on that server will also shut down. They employ NuSMV as the model checker with CTL policies. Majumdar et al. [MJM<sup>+</sup>16] propose a system for the proactive verification of management operations in Cloud and virtualized infrastructures. They precompute the N-th events that may lead to a critical event violating a policy. The system can quickly lookup if an event is critical or not, which allows the system to scale to larger infrastructures. However, their policies do not cover transitive isolation properties.

Pearson argues for accountability in cloud computing [Pea11] and highlights that tool-supported accountability is essential due to the automated and dynamic nature of infrastructure clouds. We need to automate to a large degree the monitoring and verification of dynamic virtualized infrastructures.

## Summary

Integrity verification and enforcement is important to ensure a secure foundation. However integrity verification only on the hypervisor and virtual machines covers only parts of the virtualized infrastructure. We need to assess and monitor the integrity of the entire virtualized infrastructure. Similarly, monitoring and planning changes in dynamic infrastructures, where most of the existing work focuses on networks and VM migration, covers only parts of the whole picture. In particular, we need to strive for a largely automated approach to cope with very dynamic virtualized infrastructures.



---

## 3 Automated Information Flow Analysis

In this chapter we study the automated information flow analysis of heterogeneous virtualized infrastructures. We propose an analysis system that performs a static information flow analysis based on graph traversal. The system discovers the actual configurations of diverse virtualization environments and unifies them in a graph representation. It computes the transitive closure of information flow and isolation rules over the graph and diagnoses isolation breaches from that. The system effectively reduces the analysis complexity for humans from checking the entire infrastructure to checking a few well-designed trust rules on components' information flow.

---

### 3.1 Introduction

---

The growth of IT infrastructures and the ease of machine creation have led to substantial numbers of servers being created (server sprawl). Furthermore, this led to large and complex configurations that arise by rank growth and evolution rather than by advance planning and design. Indeed, the configuration complexity often exceeds the analysis and management capabilities of human administrators. This, by itself, calls for automated security analysis of virtualized infrastructures. The high complexity of an analysis is amplified when considering security properties such as isolation, because then the analysis of individual resources must be complemented with an analysis of their composition.

In addition, virtualization providers often aim at establishing multi-tenancy, that is, the capability to host workloads from different subscribers on the same infrastructure. Also, they provide an open environment, in which arbitrary subscribers can register without trust between subscribers being justified. Therefore, we need to assume that workloads as well as VMs are under the control of an adversary, and that an adversary will use overt and covert channels in its reach.

Industry partially approaches isolation with automated management and deployment systems constraining the users' actions. However, these mechanisms can fail, lack enforcement, or be circumvented by human intervention.

---

#### 3.1.1 Contributions

---

The goal of this chapter is to study automated information flow analysis for large-scale heterogeneous virtualized infrastructures. We aim at reducing the analysis complexity for human administrators to the specification of a few well-designed trust assumptions and leave the extrapolation of these assumptions and analysis of information flow behavior to the tools.

We propose an information flow analysis tool, called *SAVE*, for virtualized infrastructures. The tool is capable of discovering and unifying the actual configurations of different virtualization systems (Xen, VMware, KVM, and IBM's PowerVM) and running a static information flow analysis based on explicitly specified trust rules. Our analysis tool models virtualized infrastructures faithfully, independent of their vendor, and is efficient in terms of absence of false negatives as well as adjustable false positive rates.

Our approach transforms the discovered configuration input into a graph representing all resources, such as virtual machines, hypervisors, physical machines, storage and network resources. The analysis machinery takes a set of graph traversal rules as additional input, which models the information flow and trust assumptions on resource types and auxiliary predicates. It checks for information flow by computing a transitive closure on an information flow graph coloring with the traversal rules as policy. From that, the tool diagnoses isolation breaches and provides refinement for a root causes analysis. The challenge of

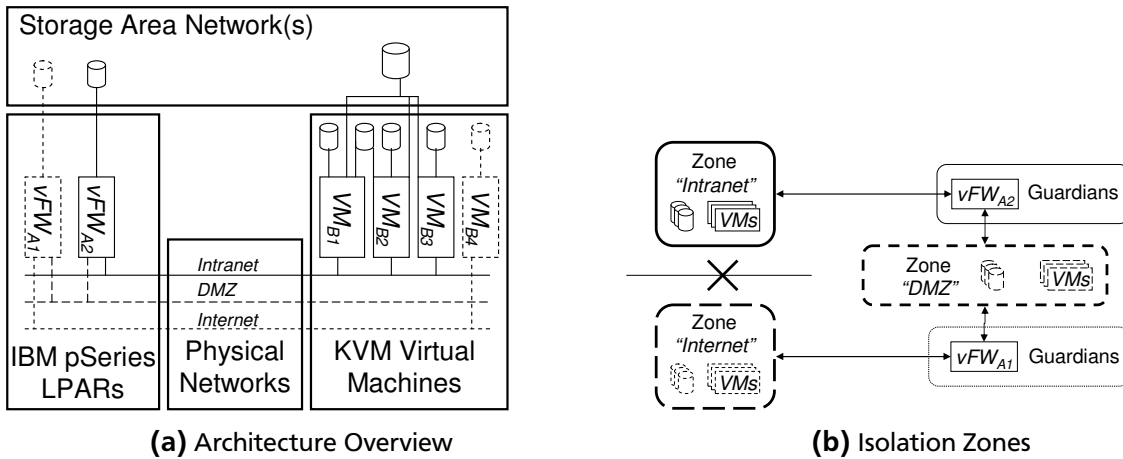
information flow analysis for virtualized infrastructures lays in the faithful and complete unified modeling of actual configurations, a layered analysis that maintains completeness and correctness through all stages, and a suitable refinement to infer the root causes for isolation breaches.

Our method applies strict over-approximation to minimize false negatives. This means that we only assume absence of flows for components that are known to isolate. This enables us to reduce the analysis correctness to the correctness of the traversal rules. As this method accepts an increase in the false positive rate, we allow administrators to fine-tune the trust assumptions with additional traversal rules and constraint predicates to obtain a suitable overall detection rate.

We report on a case study for a mid-sized infrastructure of a financial institution production environment in Section 3.6.

### 3.1.2 Applications

Our technique is applicable to the isolation analysis of complex configurations of large virtualized datacenters. Such datacenters include different types of server hardware, implementations of virtual machine monitors, as well as physical and virtual networking and storage resources.



**Figure 3.1.:** An example setup of a virtualized datacenter with an isolation policy for three virtual security zones.

Let us consider a simplified version of such a configuration in Figure 3.1a. This simplified version includes the following hardware: A IBM pSeries server, an x86 server, a virtual networking infrastructure providing VLANs, and a Storage Area Networking providing virtual storage volumes. The virtual resources (networks, storage, machines, and virtual firewalls) are depicted inside these hardware resources. Keep in mind that sizable real-world configurations contain thousands of virtual machines and tens of thousands of connections.

Figure 3.1b depicts a desired isolation topology for this example: we have three example virtual security zones “Intranet”, “DMZ”, and “Internet”. Furthermore, we permit communication between Intranet and DMZ that is mediated by a trusted guardian, such as a virtual firewall  $vFW_{A2}$ . Similarly, firewall  $vFW_{A1}$  moderates and restricts the communication between the DMZ and Internet zones, respectively. The isolation analysis must check that there do not exist components that connect two zones or are shared by two zones, while not being trusted to sufficiently mediate information flows.

Note that we focus on validating the virtualized infrastructure’s configuration. Once we have guaranteed that no undesired information flow exists except through the specified guardians, we would need to employ techniques from firewall filtering analysis, e.g. [MK05, MWZ00, Woo01], to ensure that the guardians have been configured correctly.

---

## 3.2 A Model for Isolation Analysis

---

In our work, we consider *overt* and *covert* channels. *Covert channels* [Lam73] are hidden channels that are not intended for information transfer at all, yet seem to be a common phenomenon in virtualized infrastructures [RTSS09] due to shared physical resources [WL06]. Requiring the absence of all covert channels from hypervisors, physical hosts and resources will render many resulting system impractical. Therefore, we allow administrators to capture their risk decisions and trust assumptions with regard to covert channels as part of user-configurable rules.

In the quest for a suitable requirements definition, we review information flow types [Lam73, GM82, Rus92, Man01, HY86, Rus82, KF91, Jac90] in more detail in Section 3.2.1. At this point, we note that *channel control* [Rus92] captures our requirement to specify *exceptions* to the general zoning requirements. Thus, we introduce a property we call *structural information control* that essentially lifts channel control to topology:

**Definition 1** (Structural Information Control). *A security zone is a set of system nodes and a unique color [Rus82]. A static system topology provides structural information control with respect to a set of information flow assumptions on system nodes if there does not exist an inter-zone information flow unless mediated by a trusted system node called a guardian.*

Observe that we aim at the detection of isolation breaches (information flow traces), which renders our approach loosely similar to model checking, and not at the verification of absence of information flow, which would be similar to theorem proving.

---

### 3.2.1 Flow Types

---

Information flow analysis of multi-tenant configurations in virtualized environments analyzes *overt* and *covert* channels.<sup>1</sup> An *overt channel* is intended for communication; a principal can read or write on that channel within the limits of some access control policy.

Lampson [Lam73] introduced the term *covert channel* as a channel not intended for information transfer at all. Consider a malware in VM Alice which attempts to transfer information to another instance of the malware in VM Bob, both hosted on the same hypervisor. The malware on VM Alice can, for instance, monopolize a resource<sup>2</sup> to transmit a bit observed by the malware on VM Bob in performance or throughput decrease. Similar methods combined with external observation of an honest VM's performance can determine co-location [RTSS09]. We also consider side channels as a form of covert channels where the sender and receiver are not colluding, i.e., the receiver can obtain information from the sender without the sender's knowledge. Side channels have also been exploited in virtualized infrastructures [ZJRR12]. We perceive covert channels to be a common phenomenon in virtualized infrastructures. Requiring the absence of all covert channels from hypervisors, physical hosts and resources, will render many resulting system impractical. Therefore, we allow administrators to specify a set of covert channel information flow as tolerable.

---

#### 3.2.1.1 Requirement Definition

---

Our main requirements with regard to information flow is to express isolation between security zones and exceptions to strict isolation when mediated by a trusted guardian, such as a firewall. We informally stated our security goal as *isolation* between zones, which strictly enforced corresponds to *non-interference* [GM82, Gra91]. This requirement enforces that actions in one zone do not have any effect on subsequent behavior or outputs in another zone.

---

<sup>1</sup> This is similar to the analysis of explicit and implicit information flow on high and low variables [SM03].

<sup>2</sup> Examples include launching expensive computations, flooding a cache, sending many network packets.

The transitivity of non-interference renders it, however, unsuitable to model our setting, in which information flow via guardians may be permitted, whereas the corresponding direct flow is disallowed. Agreeing to the arguments of Rushby [Rus92] and Mantel [Man01], we would need *intransitive non-interference* [HY86, Rus92] to start with. Whereas in transitive non-interference any action in a sequence is forbidden that is interfering with another isolated domain. In the intransitive case, a sequence of actions between isolated domains is allowed as long as the sequence contains only adjacent actions between non-isolating domains. In other words, *direct* information flow between isolated domains is forbidden, but flows via transitive non-isolating domains is allowed.

Another candidate is the analysis for *separation*, e.g. [Rus81, Rus82, KF91]: one removes all guardians from the system and verifies that the remaining parts are perfectly separated: “if we cut the communication channels that are allowed, then, provided there are no illicit channels present, the components of the system will become completely isolated from one another.” [Rus81] However this approach was criticized by Jacob [Jac90], because hidden channels that depend on known channels are not detected.

The concept of *intransitive non-interference* [Rus92] as well as channel control capture our requirement to allow information flows via trusted guardians or processes. For instance, two zones should not communicate with each other *unless* a guardian mediates and filters the communication. In our case, however, we are not studying single channels, but a complex topology of channels with different forms of guardians, where channels are captured through user-defined rules.

---

## 3.2.2 Modeling Isolation

---

### 3.2.2.1 Modeling Configurations

---

Our static information flow analysis is graph-based. Each element of a virtualization configuration is represented by (at least) one vertex (VMs, VM hosts, virtual storage, virtual network). Connections between elements are represented by edges in the graph and model *potential* information flow. Note that our approach requires completeness of the edges: While not all edges may later actually constitute information flows, we require that all relations that allow information flow are actually modeled as an edge. The vertices of the graph are typed: our model distinguishes VM nodes, VM host nodes, storage and network nodes, etc.

**Definition 2** (Graph Model). *Let  $\mathbb{T}$  be a set of vertex types,  $\Sigma$  an alphanumeric alphabet where  $\mathbb{A} \subset \Sigma^+$  is a set of vertex attribute names and  $\mathbb{D} \subset \Sigma^*$  is a set of attribute values. The virtualization graph model  $G = (V, E, P)$  contains a set of uniquely labeled and typed vertices  $V \subset \mathbb{V} := (\Sigma^+ \times \mathbb{T})$ , a set of edges  $E \subseteq (V \times V)$ , and a vertex properties set  $P \subset \mathbb{P} := (\mathbb{A} \times \mathbb{D})$ . A vertex  $v$  is a tuple of vertex label and type  $(l, t) \in V$ . An edge  $e$  is a pair of start and end vertices  $(v_i, v_j) \in E$ . A partial function  $\text{attr} : (\mathbb{V} \times \mathbb{A}) \rightarrow \mathbb{D}$  is defined as an attribute function which returns for a given vertex and attribute name the attribute value. The types  $\mathbb{T}$  form a type hierarchy with the root node type *Any*.*

We represent complex structures of the virtualization infrastructure by sub-graphs of multiple vertices. For instance, we construct guardians such as firewalls with complex information flow rules by a firewall vertex connected to multiple port vertices.

Information is output at one or more *information source* nodes, propagates according to *traversal rules* along the nodes and edges of the graph, and is consumed at an *information sink*.

**Definition 3** (Information Sources and Sinks). *For a set of vertices  $V$ , we define a set of information sources  $\hat{V} \subseteq V$  and a set of information sinks  $\check{V} \subseteq V$ . A vertex  $\hat{v} \in \hat{V}$  is called information source, a vertex  $\check{v} \in \check{V}$  information sink.*

### 3.2.2.2 Modeling Information Flow Assumptions

A *traversal rule* models an assumption on information flow from one vertex type to another vertex type. For instance, a traversal rule will specify that if a VM host is connected to a storage provider, this edge constitutes a direct information flow and is to be traversed. Also, a traversal rule may specify that if two VMs are connected to the same VM host, this implies the risk of covert channel communication and, therefore, constitutes an information flow.

**Definition 4** (Traversal Rules). *For the set of vertex types  $\mathbb{T}$  and the powerset  $\mathcal{P}(\mathbb{P})$  of vertex properties  $\mathbb{P}$ , the traversal rules are a propositional function of source type  $t_s$ , destination type  $t_d$ , source and destination properties subsets  $P_s$  and  $P_d$ , over a type relation  $T \subset (\mathbb{T} \times \mathbb{T})$ , and a predicate  $p$  on properties subsets:*

$$f_{\mathbb{T},\mathbb{P}} : (\mathbb{T} \times \mathbb{T} \times \mathcal{P}(\mathbb{P}) \times \mathcal{P}(\mathbb{P})) \rightarrow \{\text{stop}, \text{follow}\} :$$

$$f_{\mathbb{T},\mathbb{P}}(t_s, t_d, P_s, P_d) := \begin{cases} \text{follow} & \text{if } (t_s, t_d) \in T \wedge p(P_s, P_d) \\ \text{stop} & \text{if } (t_s, t_d) \notin T \vee \neg p(P_s, P_d) \end{cases}$$

We call traversal rules simple, if  $p$  is always true.

Similar to the tainted variable method for static information flow analysis, we employ the metaphor of color propagation [Rus82]. We associate colors to information sources  $\hat{v} \in \hat{V}$  and to vertices that have received information flow from a certain source by the evaluation of traversal rules  $f_{\mathbb{T},\mathbb{P}}$ . The total information flow of a system is the *transitive closure* of the graph traversal governed by the traversal rules  $f_{\mathbb{T},\mathbb{P}}$ . This means, that the information flow from any source to any sink can be efficiently statically analyzed by a reachability analysis between source and sink. We define graph coloring recursively.

**Definition 5** (Graph Coloring). *Let  $C$  be a set of graph colors, a function  $\text{colors} : V \rightarrow \mathcal{P}(C)$  mapping from vertices to subsets of colors. The traversal rules  $f_{\mathbb{T},\mathbb{P}}$ , graph  $(V, E, P)$ , and information sources  $\hat{V} \subseteq V$  are given. An information source  $\hat{v} \in \hat{V}$  is colored with colors  $C' = \text{colors}(\hat{v})$ . A typed vertex  $v_d \in V$  is newly colored with colors  $C'$  iff 1) there exists an edge  $e = (v_s, v_d) \in E$ , 2)  $v_s$  is colored with  $C'$ , and 3)  $f_{\mathbb{T},\mathbb{P}}(t_s, t_d, P_s, P_d) = \text{follow}$  for  $v_s = (\cdot, t_s)$  and property set  $P_s$  as well as  $v_d = (\cdot, t_d)$  with  $P_d$ . The colors of vertex  $v_d$  are the union of existing colors and new colors:  $\text{colors}(v_d) := \text{colors}(v_d) \cup C'$ . The default colors set of a vertex  $v \in (V \setminus \hat{V})$  is  $\text{colors}(v) := \emptyset$ .*

### 3.3 Isolation Analysis of Virtual Infrastructures

We apply the foundations from the preceding section to virtualized infrastructures. Our approach (see Figure 3.2) consists of four steps organized into two phases: 1) building a graph model from platform-specific configuration information and 2) analyzing the resulting model. The graph model is formally defined in Def. 2.

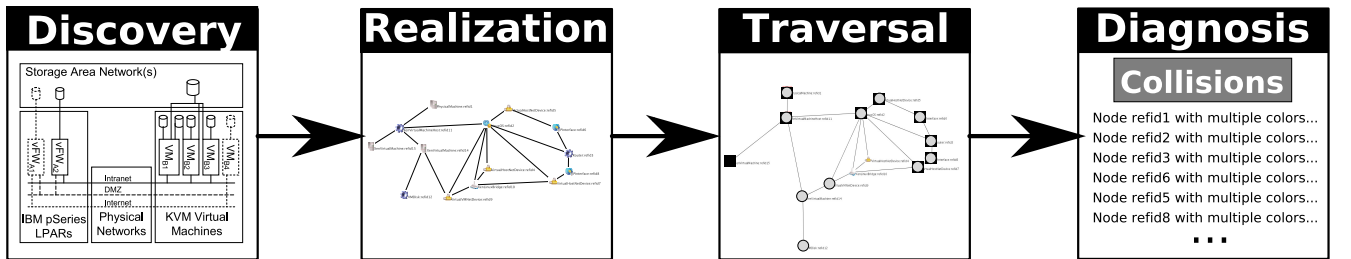


Figure 3.2.: Overview over the analysis flow.

The first phase of building a graph model is realized using a discovery step that extracts configuration information from heterogeneous virtualized systems, and a translation step that unifies the configuration

---

aspects in one graph model. For the subsequent analysis, we apply the graph coloring algorithm defined in Def. 5 parametrized by a set of traversal rules and a zone definition. The assessment of the resulted colored graph model enables a diagnosis of the virtualized infrastructure with respect to isolation breaches.

---

### 3.3.1 Discovery

---

The goal of the discovery phase is to retrieve sufficient information about the configuration of the target virtualized infrastructure. To this end, platform-specific data is obtained through APIs such as VMware VI, XenAPI, or libVirt, and then aggregated in one discovery XML file. The target virtualized infrastructure, for which we will discover its configuration, is specified either as a set of individual physical machines and their IP addresses, or as one management host that is responsible for the infrastructure (in the case of VMware's vCenter or IBM pSeries's HMC). Additionally, associated API or login credentials need to be specified.

For each physical or management host given in the infrastructure specification, we will employ a set of discovery probes that are able to gather different aspects of the configuration. We realized multiple hypervisor-specific probes for Xen, VMware, IBM's PowerVM, and LibVirt. Furthermore, if the management VM is running Linux, we also employ probes for obtaining Linux-specific configuration information. Currently, we do not discover the configuration of the physical network infrastructure. However, the framework can easily be extended beyond the existing probes or use configuration data from a third-party source.

The configuration output produced by the discovery probes consists on one hand of the physical servers that act as hypervisors with the physical devices that they contain. On the other hand the output capture the virtual infrastructure of compute, network, and storage. In particular the virtual machines with their configuration and virtual devices, the virtual network configuration with virtual switches and VLAN configurations, as well as the virtual storage that consists of filesystem stores or block devices.

---

### 3.3.2 Translation into a Graph Model

---

We translate the discovered platform-specific configuration into a unified graph representation of the virtualized infrastructure, the *realization model*. The realization model is an instance of the graph model defined in Def. 2. It expresses the low-level configuration of the various virtualization systems and includes the physical machine, virtual machine, storage, and network details as vertices.

We generate the realization model by a translation of the platform-specific discovery data. This is done by so-called *mapping rules* that obtain platform-specific configuration data and output elements of our cross-platform realization model. Our tool then stitches these fragments from different probes into a unified model that embodies the fabric of the entire virtualization infrastructure and configuration. For all realization model types, we have a mapping rule that maps hypervisor-specific configuration entries to the unified type and, therefore, establishes a node in the realization model graph. We obtain a complete iteration of elements of these types as graph nodes. The mapping rules also establish the edges in the realization model.

This approach obtains a complete graph with respect to realization model types. Observe that configuration entries that are not related to realization model types are not represented in the graph. This may introduce false negatives if there exist unknown devices that yield further information flow edges. However, we either explicitly translate configuration entries, explicitly ignore an entry, or throw a warning for any configuration entries that has not been handled explicitly. We discuss this method and its impact on the analysis detection rates in Section 3.4.1.

We illustrate this process for a VMware discovery. Each mapping rule embodies knowledge of VMware's ontology of virtualized resources to configuration names, for instance, that VMware calls storage configuration entries `storageDevice`. We have a mapping rule that maps VMware-specific configuration

entries to the unified type and, therefore, establishes a node in the realization model graph. We obtain a complete iteration of elements of these types as graph nodes. The mapping rules also establish the edges in the realization model. In the VMware case, the edges are encoded implicitly by XML hierarchy (for instance, that a VM is part of a physical host) as well as explicitly by Managed Object References (MOR). The mapping rules establish edges in the realization model for all hierarchy-links and for all MOR-links between configuration entries for realization model types.

---

### 3.3.3 Coloring through Graph Traversal

---

The graph traversal phase obtains a realization model and a set of information source vertices with their designated colors as input. According to Def. 5, the graph coloring outputs a colored realization model, where a color is added to a node if permitted by an appropriate traversal rule. We use the following three types of traversal rules (see Def. 4 and the definition of traversal rules below) that are stored in a ordered list. We apply a first-matching algorithm to select the appropriate traversal rule for a given pair of vertices. *Flow rules* model the knowledge that information can flow from one type of node to another if an edge exists. For example, a VM can send information onto a connected network. These rules model the “follow” of Def. 4. *Isolation rules* model the knowledge that certain edges between trusted nodes do not allow information flow. For example, a trusted firewall is known to isolate, i.e., information does not flow from the firewall into the network. These rules model the “stop” of Def. 4. *Default rule* means that ideally, either isolation or else flow rules should exist for all pairs of types and all conditions, that is, for any edge and any two types, the *explicit* traversal rules should determine whether this combination allows or disallows flow. In practice, the administrator may lack knowledge for certain types. As a consequence, we included a default rule as *completion*. Here, we establish a *default* flow rule: whenever two types are not covered by an isolation or flow rule, then we default to “follow”. To be on the safe side, i.e., reducing false negatives, we assume that flow is possible along this unknown type of edges.

Given this set of rules, we then traverse the realization model by applying the set of traversal rules and color the graph according to information flows from a given source. The traversal starts from the information sources and computes the transitive closure over the traversal rule application to the graph. We capture the concept of trusted guardians that mediate information flow between security zones through explicit stop rules, thereby enforcing separation, or through follow rules with sub-colors tagging the guardian flows.

---

### 3.3.4 The Coloring Traversal Rules

---

The graph coloring algorithm requires a set of traversal rules that model information flows, isolation properties, and trust assumptions. We extend the definition of our traversal rules from Def. 4 with directionality, color transformation, as well as concrete Realization model vertex types. We will propose a set of rules and explain their purposes, and leave the correctness argument to the security analysis in Section 3.4.3.

**Definition 6** (Directed and Color-Transforming Traversal Rule). *Let  $F$  be a set of follow types {stop, follow},  $\mathbb{T}$  be a set of realization model types {Port, NetworkSwitch, PhysicalSwitch, ManagementOS, PhysicalDevice, VirtualMachine, VirtualMachineHost, StorageController, PhysicalDisk, FileSystem, File, node},  $T \in (\mathbb{T} \times \mathbb{T})$  a type relation, and  $D$  be a set of flow directions  $\{\rightarrow, \leftarrow, \leftrightarrow\}$ , where  $\rightarrow$  and  $\leftarrow$  denote a unidirectional, and  $\leftrightarrow$  a bi-directional flow. A traversal rule is a tuple  $(f, t_s, t_d, d, p, g)$  with  $f \in F$ ,  $(t_s, t_d) \in T$ ,  $d \in D$ ,  $p$  is a predicate over properties and colors of the vertices, and a colors modification function  $g : \mathcal{P}(C) \times \mathcal{P}(\mathbb{P}) \times \mathcal{P}(\mathbb{P}) \rightarrow \mathcal{P}(C)$ . The color modification function operates on an input color set as well as source and destination property sets, and produces an output color set. During graph coloring (see Def. 5), a vertex  $v_d$  is newly colored with  $C_d = \text{colors}(v_s)$ . With color modifications,  $v_d$  is newly colored with  $C_g = g(P_s, P_d, C_d)$  such that  $\text{colors}(v_d) := \text{colors}(v_d) \cup C_g$ . By default  $g$  is the identity function on the input colors set.*

**Table 3.1.: Traversal Rules**

#	Type	Flow	Condition +Color Modification
<b>Trust Rules</b>			
1	stop	<i>PhysicalSwitch</i> → <i>Port</i>	Has any <i>vlan</i> color
2	stop	<i>ManagementOS</i> ↔ <i>any</i>	
3	stop	<i>PhysicalMachine</i> ↔ <i>PhysicalDevice</i>	
4	stop	<i>VirtualMachine</i> ↔ <i>VirtualMachineHost</i>	
<b>Network Switches</b>			
5	stop	<i>Port</i> ↔ <i>NetworkSwitch</i>	Port is disabled
<b>VLAN</b>			
6	follow	<i>Port</i> → <i>NetworkSwitch</i>	Port has VLAN tagging with tag \$VLAN + Create <i>vlan-\$VLAN</i>
7	follow	<i>NetworkSwitch</i> → <i>Port</i>	Port's VLAN tag matches color's one + Remove <i>vlan-\$VLAN</i>
8	follow	<i>NetworkSwitch</i> → <i>Port</i>	Port is trunked
9	stop	<i>NetworkSwitch</i> → <i>Port</i>	Port's VLAN tag mismatches color's one
10	stop	<i>NetworkSwitch</i> → <i>Port</i>	Has any <i>vlan</i> color
<b>Storage</b>			
11	stop	<i>StorageController</i> → <i>PhysicalDisk</i>	
12	stop	<i>FileSystem</i> → <i>File</i>	
<b>Default</b>			
13	follow	<i>any</i> ↔ <i>any</i>	

The traversal rules specified in Table 3.1 are a ordered list of rules. In case the condition is left empty, a *true* predicate is assumed, and in case the color modification is empty, *g* is the identity function.

**Definition 7** (Matching Rule). *Given a traversal rule  $(f, t_s, t_d, d, p, g)$  as defined in Def. 6 and a source and destination vertex from the graph traversal:  $v_s$  and  $v_d$  respectively. The rule matches iff i)  $(type(v_s) \leq t_s) \wedge (type(v_d) \leq t_d)$  where  $type(v)$  denotes the type of a given vertex  $v$  and  $\leq$  a sub-type relation, ii)  $d \in \{\rightarrow, \leftrightarrow\}$ , iii)  $p(P_s, P_d, colors(v_s), colors(v_d)) = true$  with the property subsets  $P_s$  and  $P_d$  of  $v_s$  and  $v_d$  respectively.*

The first-matching algorithm iterates over the ordered list of traversal rules and applies the matching rule defined in Def. 7. If the matching evaluates to true, the iteration stops and the matched rule is returned. The matching of the traversal rules induces a function representation of the traversal rules as defined in Def. 4, i.e., a stop or follow is returned by the matching function for a given rule tuple.

### 3.3.5 Case-Study Traversal Rules

The coloring traversal rules of the case study are listed in Table 3.1. Our trust assumptions are specified in the rules Rule 1, Rule 2, Rule 3, and Rule 4. These model that VLANs are isolated on physical switches, that the privilege VM and the physical machine are trusted and do not leak information, and that we exclude cross-VM covert channels (see Section 3.4.3).

Rule 5 simply stops an information flow if a network port is disabled. Rule 6 and Rule 7 model the VLAN en- and de-capsulation of network traffic. We refer to Section 2.1.1 for more information on network virtualization and in particular VLANs. Traffic with a VLAN tag is modeled as a new color *vlan* with the VLAN tag appended, which is created in case of encapsulation and removed in case of decapsulation. This



---

models the traffic encapsulation that is performed by VLANs and other network virtualization methods. In the case of VMware, the VLAN tag for a VM is modeled as a non-zero *defaultVLAN* property of the port. Rule 8 specifies that if a port is marked as trunked, which is required in the case of VMware to allow traffic from the VMs to the physical network interface, the VLAN traffic is also allowed to flow. Otherwise, if the *vlan* color tag mismatches the port's VLAN tag, we isolate and stop the information flow (see Rule 9). This also applies to Rule 10, which is the default isolation rule for VLAN traffic, if one of the previous rules did not match.

On the storage side, we model the behavior of the storage controller not to leak information from one disk to another with Rule 11. Furthermore, the filesystem will not leak information from one file to another (Rule 12).

The default rule Rule 13 allows any information flow that was not handled by a previous rule due to the first-matching algorithm.

We make three observations about the traversal rules: *First*, administrators can modify existing and specify further traversal rules, for instance, to relax trust assumptions or to model known behavior of specific components. *Second*, traversal rules serve as generic interface to include analysis results of other information flow tools into the topology analysis (e.g., firewall information flow analysis). *Third*, the behavior of explicit guardians (see Def. 1) is introduced by traversal rules specific to these nodes. For instance, the guardians in the exemplary Figure 3.1, Section 3.1.2, would receive a stop-rule.

---

### 3.3.6 Detecting Undesired Information Flows

---

The goal of the detection phase is to produce meaningful outputs for system administrators. For detecting undesired information flows, we color a set of information sources that mark types of critical information that must not leak. The idea of the color spill method is to introduce nodes called 'sinks' (see Def. 3). Each sink is colored with a subset of the colors corresponding to the information that it is allowed to receive. In practice, the administrator provides a list of clusters or zones that shall be isolated, and we add/mark sources and sinks according to the isolation policy with respect to these zones. In our example from Figure 3.1, Section 3.1.2, we would mark nodes from the zones "Intranet" ( $VM_{B1}, VM_{B2}, VM_{B3}$ ) and "Internet" ( $VM_{B4}$ ) as sources, and the guardians and nodes of the opposite zones ( $vFW_{A1}, vFW_{A2}$ ) as corresponding sinks, to determine isolation breaches in both directions. After the transitive closure of the traversal rules, we check whether any additional colors "spilled" into a given sink. If a sink gets connected to an additional color, then we have found a potential isolation breach. You could imagine the dedicated color sinks as a honey pot, waiting for colors from other zones to spill over.

Observe that the detection of a color spill only indicates the existence of a breach and between which zones (source-sink pairs) it has occurred. The color flow can be visualized and of some use for administrators to fix the problem. In addition, we study different refinement methods for root-cause analysis, in order to pinpoint critical edges responsible for the information flow in a industry case study (Section 3.6).

---

## 3.4 Security Analysis

---

In this section we analyze the different phases of our analysis framework with regard to detection rates. For the discovery and translation phases we discuss different fault cases, where the produced graph model differs from the actual infrastructure's configuration and topology, and how they impact the detection rates. Further we analyze how our method mitigates such faults. For the graph coloring and analysis phase we reduce the correctness to the traversal rules and discuss the rules of our case study.

---

### 3.4.1 Configuration Discovery and Translation

---

We require that the configuration discovery of virtualized infrastructures contains all elements that might solicit or prevent information flow (cf. Section 3.3.1). Further, we require that the translation modules of

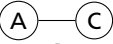
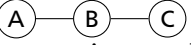
---

concrete systems are capable to correctly translate configuration elements to vertices and edges in the realization model (cf. Section 3.3.2).

In order to analyze our discovery and translation, we consider the following fault cases where the produced graph model differs from the actual virtualized infrastructure topology.

- *Extra Vertex*: The produced graph model contains a vertex that does not correspond to an element in the actual virtualized infrastructure of the mapped type.
- *Missing Vertex*: An element in the actual infrastructure is not represented by a vertex in the graph model of the corresponding type.
- *Extra Edge*: An edge is produced that does not represent an actual relation in the infrastructure topology.
- *Missing Edge*: An edge is missing in the graph model and does not represent a relation of the infrastructure topology.
- *Incorrect Vertex Attribute*: A vertex attribute can have a wrong value, where the attribute values of the model and the configuration differ. An attribute could not be set, although the attribute is present in the actual configuration. The attribute is set in the model, but not present in the actual infrastructure.

In general, the impact of the fault cases on the detection rates of our analysis is the following. An extra vertex without any additional edges has no impact on the analysis, since graph coloring requires an edge for color propagation. An extra edge may increase the false positive rate, because an additional path of color propagation can be established with the extra edge. However, an extra edge does not increase the false negative rate, because an extra edge cannot remove an existing color propagation path.

The combination of an extra vertex with a modified edge set can lead to an increase in false negatives, for instance, for the following example where the extra vertex becomes an intermediate node. The correct graph model looks like:  but an intermediate node B has been created due to a faulty discovery or translation: . This intermediate node may now also increase the false negative rate, because color propagation can be stopped at B but would have propagated between A and C directly. It is important that the direct edge between A and C has been removed, which therefore also blends in the missing edge fault case.

Missing elements can increase the false negative rate if a follow rule is not applied due to missing elements, or false positives if a stop rule is not applied. Incorrect vertex attributes can affect the predicate outcome of traversal rules. Depending on the traversal decision of an affected rule (stop or follow) this can increase the false negative (follow rule) or false positive (stop rule) rate.

### **Fault Analysis of Discovery**

Our discovery needs to mitigate the defined faults in order to not impact the detection rate of our analysis. The essential properties of our discovery are: i) complete and unmodified view of the configuration by the management system or hypervisor, and ii) complete configuration extraction.

The configuration view provided by the management hosts or hypervisors is critical for our discovery phase. We assume that the software on the configuration endpoints, i.e, the management host or hypervisor, is correct and reflects the actual state of virtualized infrastructure configuration. We assume that the software has not been compromised, which could result in a modified or incomplete view of the configuration. In addition the communication channel between the configuration endpoint and our discovery software needs to be authenticated and integrity protected. Otherwise the configuration view could be modified on the network level or provided by a malicious configuration endpoint. In practice, the trusted communication channel is realized with either HTTPS and server certificate verification, or SSH and host key verification.

---

In order to ensure a complete extraction, our framework needs to know all the existing management hosts and hypervisors from which to extract their configuration. This must be provided by the administrator. The configuration endpoints typically have access control policies in place and our discovery requires read-only access to the entire configuration. Otherwise, we may not obtain a complete configuration view.

Configuration extraction method by our discovery is to unconditionally extract the entire configuration and only in the translation phase explicitly ignore configuration elements. In the case of API-based discovery probes, we perform a complete iteration over the configuration elements (cf. Section 3.5.1 for VMware). We iterate over all elements and serialize them into XML without any further processing or modifications by the discovery probe. In the case of SSH-based probes, we either execute commands that already return XML, such as for libvirt, and which do not require further processing. Otherwise, we execute commands and parse their output, for instance for pSeries. The command output parsing will provide warnings in case of parsing errors and does not silently ignore those errors, which could result in an incomplete or incorrect configuration extraction. Furthermore, the command output is typically in a delimiter-separated format that can be parsed robustly.

A complete view by the configuration endpoint and an unconditional and complete configuration extract ensures that the configuration output is not missing elements and mitigates the faults of missing vertex/edge. The discovery does not create new configuration elements while performing the configuration extraction, therefore no extra vertex nor extra edge faults. We minimize the processing of configuration output in order to mitigate the problem of incorrect vertex attributes.

### **Fault Analysis of Translation**

The translation phase is building up on the complete and unmodified configuration output from the discovery phase as discussed earlier.

We assume we have a correct meta model that defines the vertex types and the type relations. In a practical implementation, we can use a statically typed language, such as Java or Scala in our case, that allows us to define the vertex types and their required vertex attributes, including modeling relations as attributes. Proving the correctness of the meta model itself is not feasible, but the meta model has been developed based on domain knowledge of virtualized infrastructure configurations, analyzing the configuration output of different systems and identifying common elements, and evaluating and evolving the meta model in case studies.

The translation needs to iterate over all the configuration elements and either map them to vertex types and their attributes in the model or explicitly ignore them, because an element is not relevant for our analysis and not represented in the meta model. All other configurations elements that have not been mapped nor explicitly ignored will generate a warning, because this could lead to a potentially missing vertex in the graph model. The mapping from configuration element to vertex type is a simple one-to-one mapping, e.g., a VMware VM object is mapped to a VM type in the graph model. When a configuration element has been mapped to a vertex type in the meta model, the translation will instantiate a new vertex of this type. The required attributes of the vertex are populated with the values from the corresponding attributes in the configuration element with minimal amount of processing. Enforcing the population of required vertex attributes mitigates the fault of unset vertex attributes.

Relations are either represented in the configuration output as explicit relations with an internal identifier to another configuration element or as nested configuration elements. We translate those cases explicitly into edges in our graph model. In a translation of a configuration element with internal identifier relation, both the current element and the target element of the relation are translated. An edge is then created between the two vertices. In the case of nested configuration elements, we perform a recursive translation on the nested elements and create edges between a vertex and the produced vertices of the recursive translation.

---

## 3.4.2 Graph Coloring and Information Flows

---

The correctness of the graph coloring builds upon the faithful representation of the virtualized infrastructure configuration and topology by the graph model as discussed previously. We now show that the graph coloring and detection of undesired information flows reduces to the correctness of the traversal rules and the coloring of sources and sinks.

In Section 3.3.6 we introduced the detection of undesired information flows as “color spills”. More formally, a color spill is an alarm event  $A$  for which there exists a vertex  $\check{v}$  with an allowed color set  $\check{C}$  and  $\check{v}$  has been colored with a color  $c$  for which  $c \notin \check{C}$ .

In order for an  $A$  event to be raised, there must exist an information source  $\hat{v} \in \hat{V}$  with  $\hat{C} = \text{colors}(\hat{v})$  and a color propagation path that colors  $\check{v}$  with  $c \in \hat{C} \cup \check{C}$ . We show that such a color propagation path exists with induction over the length  $n$  back-trace graph traversal. The induction aligns with the recursive definition of the graph coloring (Def. 5).

Initialize a set  $\vec{E} = \emptyset$  which holds our color propagation path.

**Induction start  $n = 1$ :** the sink  $\check{v}$  is colored with  $c$  because of the alarm event  $A$ .

**Induction step  $n + 1$ :** A colored vertex  $v_n$  could only have been colored with  $c$  if

- (a)  $v_n$  is source  $\hat{v}$  with the corresponding color  $c \in \hat{C}$  (then we are done and output  $\vec{E}$ ) **or**
- (b) there exists an edge  $e = (v_{n+1}, v_n)$  with  $v_{n+1} = (\cdot, t_{n+1}, p_{n+1})$  and  $v_n = (\cdot, t_n, p_n)$  for which holds: the traversal rules  $f_{\mathbb{T}, \mathbb{P}}(t_{n+1}, t_n, p_{n+1}, p_n)$  evaluate to follow, and either  $v_{n+1}$  is colored with  $c$  or the traversal rules’ color transformation produces  $c \in g(\text{colors}(v_{n+1}))$ . Accumulate  $\vec{E} := \vec{E} \cup \{e\}$ .

Assuming that our graph model is correct and contains all the required vertices, vertex types, edges, and attributes, the recursive color propagation is only further influenced by the traversal rules. Thereby the correctness of a color spill alarm event  $A$  reduces to the correctness of the traversal rules and their flow decision.

Furthermore, as part of the security policy configuration, the administrator needs to assign colors to the information sources and sinks, which establishes the isolation policy.

---

## 3.4.3 Correctness of the given Traversal Rules

---

The correctness of the traversal rules from Table 3.1, Section 3.3.4 remains to be shown, where we need to analyze on two levels: i. correctness of individual rules and ii. correctness of their composition.

---

### 3.4.3.1 Individual Rules

---

We highlight the most important points here, followed by a detailed examination of the traversal rules.

- *Network:* We model correct implementation of switches for port disablement (Rule 5), and VLAN encapsulation and decapsulation (Rules 1, and 8, to 10).
- *Physical Machine, Hypervisor, VM Stack:* We claim secure isolation by management OS and physical machine (Rules 2 and 3) as well as cross-VM isolation (Rule 4). The former rules are elementary for virtualization security, the latter rule is arguable as it models the hypervisor’s multi-tenancy capability and needs to be reconsidered depending on the actual environment (cf. [RTSS09, Aci07] and discussion in the following).
- *Storage:* We model secure separation by physical disks as well as by the file system (Rules 11 and 12), where the latter rule is systematically enforced by virtualization vendors (e.g., [VMw06]) and can be checked automatically [YTEM06].

---

We are now discussing and examining the individual traversal rules in more detail. *First*, let us analyze the rules for network switches and VLAN traffic. Rule 5 assume a correct implementation of an isolation by network switches for switched-off ports. Rules 6 and 7 establish the VLAN en- and decapsulation by network switches and are interesting for the security analysis. The rules assign a VLAN-specific color to information flow for in-ports with VLAN tagging and only allow information traversal at out-ports with matching VLAN tags. This models the VLANs' traffic separation by encryption lifted to VLAN tags as well as a cross-session key separation assumption, standard for secure channels: messages encrypted under one VLAN tag cannot interfere with messages encrypt under other VLAN tags and can only be decrypted under the same VLAN tag. Rule 9 stops information flow at ports with non-matching VLAN tags accordingly. Rule 8 has information flow follow through for trunked VLAN ports. Otherwise, we assume that the network and physical switches securely configure and implement VLAN traffic isolation for flows from switch to port (Rules 10 and 1). We conclude that these assumptions are natural and model correct network behavior.

We are aware of attacks on VLANs, in particular VLAN hopping attacks, but virtual and physical switches mitigate these attacks. For instance, VMware vswitches do not support dynamic trunking and drop frames with double VLAN tagging, which are the cause of VLAN hopping attacks. Other switches, in particular physical ones, can be configured as well to not be vulnerable to such attacks.

*Second*, let us consider the stack of physical machine, hypervisor and VMs. Rules 2 and 3 make the assumptions that a management OS and physical host provide secure isolation and that all information flow is accounted for explicitly. These assumptions are necessary for virtualization security, as information leakage from these components can subvert the entire system's security, and model standard trust assumptions. Rule 4 is interesting as it assumes that hypervisors sufficiently separate VMs against each other, that is, that information flow through cross-VM covert channels can be neglected. Research results exist that highlight cross-VM covert channels, for instance [RTSS09, Aci07]. Therefore, this trust assumption on the hypervisor's multi-tenancy capability must be subject to thorough debate.<sup>3</sup> Whereas the isolation assumptions on physical machine and management OS are natural and well founded, we conclude that the modeling of covert channels is a key trust decision for the hypervisor model.

Research on covert channels in high-assurance micro-kernels, such as *SeL4* [KEH<sup>+</sup>09], has shown that it is possible to prove the absence of storage covert channels [MMB<sup>+</sup>13], i.e., to prove a variant of intransitive non-interference [MMB<sup>+</sup>12]. However, timing channels are more difficult and an empirical study [CGMH14] has investigated timing channels in *SeL4* and their countermeasures. Although countermeasures can be effective, newer processor architectures and optimization renders them less effective and creates new timing channels. In practice, the policy decision of cross-VM flows is based on a risk assessment by the organization and may vary depending on the hypervisor. For instance, VMware vSphere 5.0 has been certified to Common Criteria EAL4+ [VMw16] which supports the risk assessment by the security operator.

*Third*, let us consider the information model for storage. Rule 11 models that the storage controllers are capable of separating information flow to physical disks, whereas Rule 12 establishes that the file system prevents cross file information flow through its access control enforcement, which found attention in research and can be checked with tool support [YTEM06].

---

### 3.4.3.2 Traversal Rules and Detection Rates

---

The traversal rules impact the detection rates in the following way:

- *Explicit Knowledge Model*: The explicit traversal rules model all and only known facts about information flow and isolation. Thus, traversal rules focus on preventing false negatives introduced by invalid assumptions.

---

<sup>3</sup> For high-security environments, we recommend to set this rule to follow and therefore only relying on physical separation, yet dismissing hypervisor multi-tenancy.

- *Strict Over-abstractio*n: When in doubt, the traversal rules must be a conservative estimate towards information flow, that is, model a super-set of potential information flow. By that, traversal rules will never introduce false negatives at the cost of additional false positives.
- *Default-Traversal Behavior*: The default rules establishing completion on the traversal rules must all be *default-follow rules*, that is, evaluate undetermined cases to follow and log such results. Thus, the completion will only introduce false positives but never false negatives.

We conclude that the traversal rule robustness principles are all lined up to fence off false negatives, yet at the cost of false positives. Whereas this trade-off benefits a conservative security analysis, it impacts its effectiveness, as becomes manifest in its overall detection rate.

---

### 3.4.4 Traversal Rules Coverage

---

We discuss the completeness and coverage of a given traversal rules set, which influence the detection rate of our analysis.

**Definition 8** (Completeness). *For the set of vertex types  $\mathbb{T}$  and a set of vertex properties  $\mathbb{P}$ , traversal rules  $f_{\mathbb{T},\mathbb{P}}$  are called complete if  $T$ ,  $P_s$ ,  $P_d$  associated to  $f_{\mathbb{T},\mathbb{P}}$  cover all pairs of types in  $\mathbb{T}$  and all properties in  $\mathbb{P}$ . We call a default rule a completion of incomplete traversal rules  $f_{\mathbb{T},\mathbb{P}}$ , if it maps all undetermined cases to either stop or follow. We call non-default rules explicit.*

Whereas completeness is a property of a set of traversal rules, we define *coverage* as in how far a set of traversal rules determines the analysis of a graph deterministically without invoking the default rule.

**Definition 9** (Coverage). *For the set of vertex types  $\mathbb{T}$  and a set of vertex properties  $\mathbb{P}$ , consider a realization model graph  $G = (V, E, P)$  as in Def. 2 and the subset of edges  $E' \subseteq E$  that are matched by explicit traversal rules  $f_{\mathbb{T},\mathbb{P}}$ . We call the quotient of number of explicitly matched edges to total number of edges coverage:  $c = |E'| / |E|$ .*

A high coverage reflects that the operators have explicitly decided for pairs of infrastructure components if information flow is possible or not. Under the assumption of the correctness of these explicit decisions, this will decrease our false positive or negative rate. Otherwise for the implicit case, we either increase the false positive rate with a default follow decision or the false negative rate with a stop decision. Thereby, with a high coverage we reduce the number of implicit decisions and its implications on the detection rates.

The traversal rules specify general assumptions on information flow in virtualized environments and, thereby, embodies a part of the overall trust assumptions. The specification of traversal rules is therefore orthogonal to the isolation policy of a system. Whereas our system comes with a root set of traversal rules as base line trust assumptions, we allow users to specify multiple sets of *user-defined traversal rules* and thereby *user-defined trust assumptions*.

---

### 3.4.5 Discussion

---

The transitive closure over the graph coloring securely lifts the isolation analysis to an analysis of the traversal rules  $f_{\mathbb{T},\mathbb{P}}$ . Therefore, the correctness of the traversal rules becomes a make-or-break criterion for the analysis method and impacts the detection rate.

We observe a *complexity reduction*: the simple traversal rules have a complexity of their type relation  $T \subset (\mathbb{T} \times \mathbb{T})$ . In practice,  $|\mathbb{T}| \ll |V|$  as well as  $|\mathbb{T}|^2 \ll |E| \leq |V|^2$ , with the number of properties set for  $f_{\mathbb{T},\mathbb{P}}$  being small. Therefore, the complexity of analyzing the traversal rules  $f_{\mathbb{T},\mathbb{P}}$  is much smaller than the complexity of isolation analysis. This allows administrators to explicitly model and thoroughly and efficiently check their knowledge and trust assumptions about information flow and isolation.

---

Because our traversal rules base on the principle of *over-abstraction*, that is, resort to default-traversal in undetermined cases, the method excludes false negatives, at the cost of additional false positives. The method is therefore always on the conservative side, even though we are well aware that the false positive rate impacts the overall detection rate [Axe00]. We provide the general analysis framework and offer user-defined traversal rules to fine-tune the analysis method to reduce false positives and maximize the Bayesian detection rate. Also, we experiment with refinement methods for a subsequent root-cause analysis to pinpoint critical information flow edges.

In principle, our tool is in a similar situation as the first intrusion detection systems. There do not exist standardized data sets to quantify and calibrate false positive and false negative rates. We approach this situation by obtaining real-world data from third parties and are testing the analysis method in sizable real-world customer deployments, such as the case study discussed below.

Our framework analyzes a snapshot of the virtualized infrastructure configuration and topology at a given point in time. Thereby we can only detect illicit information flow paths and isolation violations in a given snapshot. Persistent information flow, e.g., through persistent storage such as a harddisk attached to a VM, may lead to isolation violations that manifest themselves over multiple configuration snapshots. For instance, a VM writes to its attached virtual harddisk, then the harddisk is detached from the VM and attached to a new VM, which then reads the information stored on the harddisk. One way to solve this problem with the approach of this chapter is to mark certain infrastructure elements as persistent, such as a virtual harddisk. Once a graph vertex of a persistent element has been colored, this vertex will become an information source in the analysis of the next snapshot. Another solution direction is to use a model that tracks the changes in the infrastructure (cf. Chapter 7) and express security policies over multiple states of the model, which we consider as future work (cf. Chapter 9).

---

## 3.5 Implementation

---

We have implemented a prototype of our automated information flow analysis in Java that consists of roughly twenty thousand lines of code. Furthermore, we have additional scripts written in Python that perform post-processing for visualization purposes and refinement for root-cause analysis. The prototype consists of two main programs, that is, the discovery, and a processing and analysis program. The result of the discovery is written into an XML file and is used as the input for the analysis.

---

### 3.5.1 Discovery

---

The functionality of the discovery and its different probes were already outlined in Section 3.3.1. There exist different ways to implement a discovery probe. A probe can establish a secure console (SSH) connection to the virtualized host or the management console where commands are executed and the output is processed. Typically, the output is either XML, which is stored in the discovery XML file directly, or the output has to be parsed and transformed into XML. As alternative to the secure console, a probe can connect to a hypervisor-specific API, such as a web service, that provides information about the infrastructure configuration.

We illustrate the discovery procedure with VMware as example. Here, the discovery probe connects to *vCenter* to extract all configuration information of the managed resources. It does so by querying the VMware API with the `searchManagedEntities()` call of the `InventoryNavigator`, which provides a complete iteration of all instances of `ManagedEntity`, a base class from which other managed objects are derived. We ensure completeness by fully serializing the entire object iteration into the discovery XML file, including all attributes.

---

## 3.5.2 Processing

---

The *processing* program consists of the transformation of the discovery XML into the realization model, the graph coloring, and the analysis of the colored realization graph.

The realization model is a class model that is used for generating Java class files. During the transformation of the XML into the realization model, instances of these classes are created, their attributes set, and linked to instances of other classes according to the mapping rules (cf. Section 3.3.2). Again, we illustrate this process for VMware. Each mapping rule embodies knowledge of VMware's ontology of virtualized resources to configuration names, for instance, that VMware calls storage configuration entries `storageDevice`. The edges are encoded implicitly by XML hierarchy (for instance, that a VM is part of a physical host) as well as explicitly by Managed Object References (MOR). The mapping rules establish edges in the realization model for all hierarchy links and for all MOR links between configuration entries for realization model types.

The traversal rules used for the graph coloring (cf. Section 3.3.3 and Section 3.3.4) are specified in XML. Intermediate results, such as the paths of the graph coloring, can be captured and used for further processing, i.e., visualization. We implemented Python scripts that generate input graphs for the *Gephi visualization framework* [BHJ09].

---

## 3.6 Case Study: Virtualized Infrastructure Isolation

---

We launched a case study with a global financial institution for a performance evaluation and for further validation of detection rates and behavior in large-scale heterogeneous environments. The analyzed virtualized infrastructure is based on VMware and consists of roughly 1,300 VMs, the corresponding realization model graph of 25,000 nodes and 30,000 edges. The production system has strong requirements on isolation between clusters of different security levels, such as high-security clusters, normal operational clusters, backup clusters and test clusters. In addition, we can work with a comprehensive inventory of virtualized resources that serves as specification of an ideal state (machine placement, zone designation and VLAN configuration) and as basis for alarm validation.

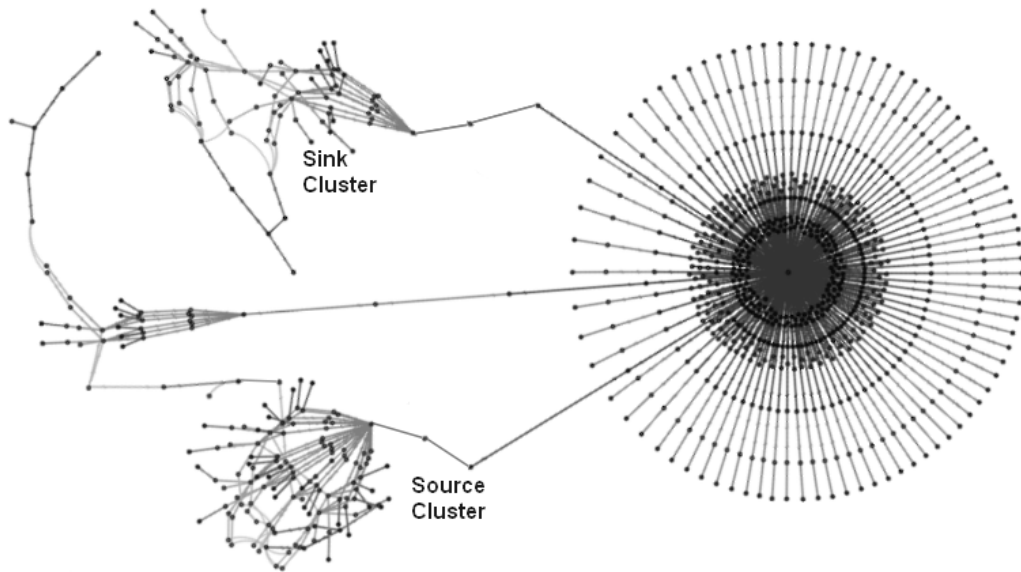
We examine preliminary lessons learned, where we first consider the operation of the tool itself. The phases discovery, transformation to realization model and graph coloring executed successfully. The visualization of all results presented a challenge as a 25,000-node/30,000-edge graph overburdened the built-in visualization of the tool.

From a performance perspective, the discovery of the infrastructure using the VMware probe in combination with vCenter requires about *seven minutes*, and results in a discovery XML file with a size of *61MB*. The discovery was performed in a production environment, where network congestion and other tools using the same vCenter can have a negative effect on the discovery performance. The overall analysis of the infrastructure using the discovery XML file requires *53 seconds*, where *46 seconds* are spent on the graph coloring. This demonstrates a reasonable performance for the discovery and analysis of a mid-sized infrastructure, such as the one in our case study.

From a security perspective, the tool indeed found several realistic isolation breaches, which we highlighted by adding extra edges between breached clusters. All isolation breaches constituted potential information flows. By that we could show actual breaches between high-security, normal operational and test clusters. We have furthermore shown that the documentation of the permitted flows was incomplete: One breach that the system identified violated the initial policy given by the customer and was fixed by augmenting the policy. In terms of detection rates, we initially had false positives due to missing stop rules in our rule set, where flows via the hypervisors devices was not explicitly stopped but where allowed by the default rule. Adding explicit stop rules increased our rule coverage and removed the false positives. Without a reference set or tool, we could not quantify the false negatives rate.

Root-cause analysis answers the question which edges and nodes are ultimately responsible for the breach. We found that color spill after a traversal to a new cluster may hamper the subsequent root-cause analysis.





**Figure 3.3.:** Root-cause analysis of a source cluster with information flow to a sink cluster. The tree refinement derives only the sub-graphs relevant for an isolation breach.

We therefore introduced multiple automated refinement mechanisms after the graph-coloring phase to support the elimination of classes of potential root causes. *First*, we benefited greatly from a process of elimination, that is, to exclude, for instance, that information has flown over storage edges. *Second*, it was helpful to allow partial coloring, in particular to stop color propagation after detecting a breach to another cluster. *Third*, we introduced a reverse flow tree that captured which path information flow took as prelude to a breach. Figure 3.3 depicts an example of such a color tree: the tree is a sub-graph highlighting a cross-cluster information flow path. *Fourth*, we further refined this tree by extracting critical edges, such as passed VLANs, to pinpoint routes of information flow.

In conclusion, we added a refinement phase driven by reusable Python scripts. We obtained multiple realistic alarms and could trace their root causes. The graph export to Gephi enabled the efficient visualization of root causes and information flows for human validation.

---

### 3.7 Summary

---

We demonstrated an analysis system that discovers the configuration of complex heterogeneous virtualized infrastructures and performs a static information flow analysis. Our approach is based on a unified graph model that represents the configuration of the virtualized infrastructure and a graph coloring algorithm that uses a set of traversal rules to specify trust assumptions and information flow properties in virtualized systems. Based on the colored graph model, the system is able to diagnose isolation breaches, which would violate the customer isolation requirements in multi-tenant datacenters. We showed in our security analysis that we can reduce the correctness and detection rate to the correctness and coverage of the graph traversal rules. Based on existing research and systems knowledge, we submit that the present traversal rules are natural and correct.

The next step is to extend our approach towards other configuration properties, such as dependability, and propose a more generalized analysis framework. In addition, dynamic analysis becomes more important with increasing size of the topology and change frequency. Our current approach performs a static analysis of a given configuration state. A dynamic analysis can be emulated by running multiple static analyses and comparing the resulting realization models. However, a truly dynamic analysis needs to analyze small configuration changes and efficiently determine their effect on the topology.



---

## 4 Virtualization Assurance Language

In this chapter we study the security requirements of virtualized infrastructures and propose a practical tool-independent policy language for security assurance. Our language proposal has a formal foundation, and allows for efficient specification of a variety of security goals. The language is well-suited for automated analysis, be it by model checking or theorem proving.

---

### 4.1 Introduction

---

The complexity of cloud configuration with respect to assuring high-level security goals is challenging. It calls either for infrastructure-wide access control and deployment mechanisms to enforce the security goals automatically or for verification mechanisms to check for breaches of the goals. In any case, we need a specification language for high-level assurance goals. Such a language plays a different role in the three cases mentioned: *First* in the access enforcement case, the security assurance language is an auxiliary input to the policy decision engine that has in turn the function to ensure that the high-level assurance goals are preserved by access requests. *Second* in the automated deployment case, the deployment mechanism establishes deployment patterns that maintain the high-level security goals. Best practices and deployment templates that incorporate some security targets are insufficient to fulfill high-level security goals for the entire topology, because a series of local configuration transitions, which fulfill a local-view security property, may still breach a topology-level security goal in a global view. *Third* in the verification case, the high-level security goals constitute the verification target, against which the actual infrastructure is evaluated.

There already exist specification languages for virtualized environments. These languages aim at provisioning (cf. [DMT10, MGHW09]), or network and reachability properties, e.g., firewall topology or distributed network access control [DDLS01]. In the former case, the specification languages are restricted to single resources, notably virtual machines, however do not have provisions for statements over the topology. In the latter case, the languages have provisions to model the topology and properties thereof, however they do not provide language primitives for expressing diverse security statements as needed in virtualized infrastructures.

We derived the following three categories of interesting security statements for virtualized infrastructures from existing research literature such as [BCP<sup>+</sup>08, OGP03, RTSS09]: operational correctness, failure resilience, and isolation. *First*, operational correctness ensures that services are correctly deployed and that their dependencies are reachable. *Second*, failure resilience ensures that the effects of single component failures cannot cascade and affect many entities. *Third*, isolation ensures that different security zones are properly separated and that traffic between security zones is only routed through trusted guardians.

The goal of this work is to study such high-level security properties of virtualized infrastructures and propose a policy language to express these as goals. We call the resulting language Virtualization Assurance Language for Isolation and Deployment (*VALID*).

---

#### 4.1.1 Contribution

---

We contribute the first formal security assurance language for virtualized infrastructure topologies. More precisely, we model such an assurance language in the tool-independent Intermediate Format (IF) [AVI03], which is well suited for automated analysis. We lay the language's formal foundations in a set-rewriting approach, commonly used in automated analysis of security protocols, with access to graph analysis

functions. In addition, we propose language primitives for a comparison of desired and actual states. As a language aiming at expressing topology-level requirements, it can express management and security requirements as promoted by [DDLS01]. Management requirements in the cloud context are, for instance, provisioning and de-provisioning of machines or establishing dependencies. Security requirements are, for instance, sufficient redundancy or isolation of tenants. To test soundness and expressiveness of our proposal, we model typical high-level security goals for virtualized infrastructures. We study the areas deployment correctness, failure resilience, and isolation, and propose exemplary definitions for respective security requirements in *VALID*. Further, we discuss the security requirements of a cloud deployment in a case study and how suitable *VALID* is to cover them.

---

## 4.1.2 Outline

---

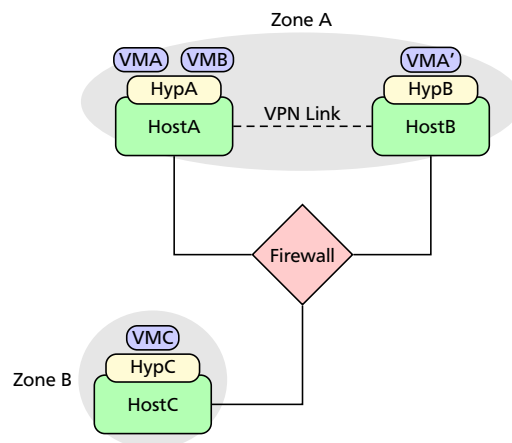
We structure this chapter in a top-down way. We first propose infrastructure-level assurance goals for virtualized systems in Section 4.2. These goals are a diverse sample of the language scope. In Section 4.3, we specify our requirements on the virtualization assurance language. We lay the language’s formal foundations in Section 4.4, that is, we introduce its roots in the Intermediate Format (IF) [AVI03] and our virtualization specific language primitives and syntax. In Section 4.5, we propose formal specifications of checkable attack states for the assurance goals defined in Section 4.2. Thereby, we exemplify the use of *VALID* in its application domain. We discuss the coverage of our language as part of a case study with the security requirements of a cloud deployment in Section 4.2.4. Finally, we conclude this chapter and briefly discuss a virtualization assurance system that would incorporate *VALID* in Section 4.7.

---

## 4.2 Virtualized Infrastructure Security Goals

---

We distilled three categories of virtualized systems security goals based on common problems described in existing research literature: *Operational Correctness*, *Failure Resilience*, and *Isolation*. Furthermore, for each of these categories we identified specific goals that our language should be capable of capture and express efficiently. Figure 4.1 depicts a simple virtualized system example that we will use to illustrate the different security goals.



**Figure 4.1.:** Virtualized Infrastructure Example Topology

---

### 4.2.1 Operational Correctness

---

Operational correctness describes that a service is both correctly deployed and reachable. It bears some similarity to the *Liveness* property introduced in [AS86, Lam77] and informally states that “good things”

---

will eventually happen for a service. Configuration mistakes often lead to unavailability of services in traditional data center environments (cf. [OGP03]) and is only intensified in virtualized environments due to their increasing complexity (cf. [BCP<sup>+</sup>08]).

### **Deployment Correctness**

Deployment correctness means that an entity is deployed in correct operational conditions, which includes multiple factors: i. The geographic location of the host system can have legal and technical consequences, e.g., conflicts with privacy laws, or long end-to-end delay due to geographic disparity. ii. Properties of the host system such as capabilities and reliability can have a significant impact on the service. iii. Furthermore, the configuration of the host system and service has to be correct that the service can actually be run on the host.

### **Reachability**

Reachability means that an entity is connected to all its operational dependencies. On one hand, these dependencies can be network reachability, i.e., the VM and physical host are actually reachable over the network from the client-side. On the other hand, these dependencies can be resource dependencies in general, e.g., that a VM is able to access services on other nodes. All such dependencies have to be fulfilled in order that the operational correctness of the service is given.

---

## **4.2.2 Failure Resilience**

---

Failures of components in a computing environment are unavoidable, but a resulting failure of services, which are visible to the end users, can be mitigated. Such containment of component failures are pointed out in [OGP03] and can be summarized as: failure compartmentalization due to *Independent failure*, and prevention of cascading failures and limitation of failure impact due to *Redundancy*.

### **Independent Failure**

Independent failure means that failures of an entity are well-contained and that dependencies of entities with the same function will fail independent from each other. This goal nurtures a diversity of the components deployed in the computing environment. A typical software stack in a virtualized system consists of a hypervisor, management operating system, virtual machine system, and the service application. A diversity in this stack, such as using different hypervisors from different vendors, will have an isolated failure in case of faults in one of these hypervisor implementations. Independent failure can be satisfied in the example scenario, in case the hypervisors *HypA* and *HypB* hosting the service VMs are provided by different vendors.

### **Redundancy**

Redundancy means that sufficient replication enforces that individual component failures will leave overall service availability unharmed. The necessary level of redundancy depends on the desired failure resilience for a service, which also depends on its criticality. Sufficient redundancy implies the absence of a single point of failure (SPoF). A SPoF exists in a system, if a dependency of a service is only satisfied by one entity in the whole system. The absence of such a SPoF entity will increase the failure resilience due to the limitation of a cascading failure effect on dependent services. In our example, the service running in *VMA* is replicated in *VMA'*, both running on different physical machine and interconnected with two independent network links. Path redundancy, i.e., fully disjoint paths, is also important and we observe in our example that only a single path is available from *VMC* to the replicated service of *VMA*.

---

### 4.2.3 Isolation

---

In virtualized environments, such as public infrastructure clouds, we see multi-tenancy in order to increase the utilization of the system. Isolation compares to *Safety* [AS86, Lam77] that undesired information flow do not happen. In [RTSS09], the problem of undesired information flow in public infrastructure clouds was exposed.

#### Isolation of Zones

Isolation of zones means that specified security zones are isolated from each other, either by correct association of machines to zones or by enforcement of flow isolation between any entity of different zones. A security zone can be any set of entities in the virtualized environment. For example, a zone in the case of tenant isolation is the set of resources used by a tenant, and zone isolation is given if the tenants do not have access to common resources. In the illustrated example, we have defined two security zones *Zone A* and *Zone B* that are disjoint, i.e., isolated of each other.

#### Guardian Mediation

Guardian mediation means that information flow between zones is allowed if, and only if, mediated by a trusted guardian. In case information flow is allowed between the two security zones defined in our example case, the *Firewall* guardian has to mediate the traffic between the zones.

#### Chinese Wall

The *Chinese Wall* isolation policy in the context of virtualization security describes that a physical host is not serving VMs of conflicting tenants. For example, a VM of *Customer A* should not be hosted on the same physical machine as a VM of competing *Customer B*. Such a policy can be implemented using the sHype [SJV<sup>+</sup>05] hypervisor. *VMA* and *VMC* in our example case are virtual machines of conflicting tenants, therefore they can not be hosted on the same physical host with regard to the Chinese wall policy. In a simplified form this policy can also be considered as a disjoint placement policy, i.e., two tenants should never be placed on the same host.

#### Secure Channels

Secure channels capture that certain information flow is only permitted over secure channels, such as provided by VLAN or VPN in terms of network resources. A VPN link is established between *HostA* and *HostB* in our example that acts as a secure channel.

---

### 4.2.4 Case Study: Policy Scope

---

We validate the scope of our policy language using a policy catalog of a telecommunications and cloud service provider. The catalog consists of roughly 50 policies where one half capture technical security aspects and the other half cover organizational security and processes. We consider the organizational security policies as out of scope for our language, since we are focusing on technical aspects as part of the infrastructure's configuration and topology, and not on aspects such as education and documentation.

We can further divide the technical policies into two groups. The first group is concerned with aspects such as (virtual) network topology and configuration, virtual machine security, and placement. These policies align with the policies that we propose and formalize in this chapter. The second group of technical policies deals with aspects such as access control, patch management, service dependencies, and hypervisor security. Either we formalize such policies also in this chapter, e.g., service dependency, or the policies are in line with the scope of our language, but not yet formalized.

---

## 4.3 Language Requirements

---

In this section we establish the requirements for our policy language. They need to capture that the language has to deal with a variety of virtualized infrastructure security goals as well as to enable the automated analysis of an infrastructure with regards to the specified policies.

### Formal Foundations

Virtualized environments can gain complexity beyond human oversight and therefore require tool-supported deployment and analysis. Thus, we expect the security assurance language to have formal rigor and be suitable for automated reasoning.

### Expressiveness

There are many different security requirements imposed on virtualized infrastructures. Therefore, we require that the security assurance language needs to be able to efficiently express a wide range of security properties as discussed in Section 4.2. *First*, the language needs to have *three expression layers*: i. statements about properties of resources, e.g., their IP address or functional classification, ii. set operations, such as membership in security zones, iii. graph operations, such as existence of an information flow or dependency path in a graph model of the topology. *Second*, the language needs to be *reflexive* and *self-contained*, that is, one can define new security goals with the existing terms of the grammar and without the need of auxiliary grammar.

We propose that the security assurance language shall express *attack states*, that is, states in which a security property is violated, as well as *ideal states*, that is, states that assure a correct system behavior. Whereas the first approach is suitable for more efficient security analysis (model checking) without complete state exploration, the second approach is suitable for complete verification (theorem proving).

### Tool and Standard Independence

Security policies in virtualized environments are a new field without settled predominant standards. We require the specification language to be independent from a specific vendor's tool or a specific standard.

### Desired State Comparison

The validation of security properties of virtualized environments provides two different views on the state of such a virtualized infrastructure: a *desired state* or the *ideal world*, as specified in the policy, and an *actual state* or *real world*, i.e., the current configuration of the virtualized infrastructure. One specific goal of our assurance language is to express comparisons of a desired state and an actual state discovered in a configuration.

In some cases it is necessary to make statements about ideal elements as well as real elements in the very same policy statement. Consider the example that a VM should be hosted on a specific host. Or in other words, the goal is breached if the VM is hosted on a different machine than specified. This breach can be efficiently captured using both elements from the ideal and real world in one policy statement. We specify that we have an ideal machine hosting the VM and also a real machine hosting the same VM. To describe the placement breach, we say that these two machines do not correspond to each other, i.e., the real machine is not the same as the ideal one in terms of the given properties. Therefore, if such a statement holds, we observed a placement breach.

---

## 4.4 Language Syntax and Semantics

---

We propose a specification and reasoning language for security properties of virtualized environments based on set-rewriting and conditions over states. Whereas future editions of this work will include dynamic aspects and therefore specify state transitions (cf. Chapter 5), we confine ourselves for the policy language to static specification of a desired state and its relation to an actual state.

**Table 4.1.:** Basic type constants for virtualized infrastructures.

Type Symbol	Description
node	denotes the superclass of types in $\mathbb{T}_N$ .
set	denotes a set of node nodes.
machine	denotes a virtual machine.
hypervisor	denotes a hypervisor on a host or VM.
host	denotes a physical host.
machineOS	denotes an operating system of a virtual machine.
hostOS	denotes an operating system of a physical host.
network	denotes a network component
zone	denotes an isolation zone of an infrastructure.
class	denotes a functional class of similar components.
guardian	denotes a trusted guardian node.

*VALID* uses a subset of the AVISPA Intermediate Format (IF) [AVI03] as its basis, a language for automated deduction based on set manipulation and conditions over state expressions. We chose IF as the basis for our work because of its capability to efficiently express goals as stated in Section 4.2, its natural extensibility to state transition formulations, its tool-independence, and its close relation to general-purpose automated deduction, which is given due to the strong formal foundation of IF, and its support by many model checkers and theorem provers.

#### 4.4.1 Terms and Types

We start from *atomic terms*, that is constants and variables. The value of a constant is fixed, e.g., the symbol for the type *machine*. We call the set of all constant terms *signature*. A variable can be matched against any value (of matching type). Atomic terms with different symbols have different values.

**Definition 10** (Term Algebra). *We define a term algebra over a signature  $\Sigma$  and a variable set  $\mathcal{V}$ . Constants and variables are disjoint alphanumeric identifiers: constants start with a lower-case letter; variables start with an upper-case letter. The signature  $\Sigma$  contains a countable number of constant symbols that represent resource names, numbers and strings.*

We typeset IF elements in sans–serif. The atomic terms are typed (see Table 4.1):

**Definition 11** (Type System). *We have a set of basic types  $\mathbb{T}$  from Table 4.1. We write  $t : \tau$  for a term  $t$  having type  $\tau$ . Variables can be untyped or typed. If a variable has a basic type, it can generally only be matched against a constant with matching type. The type symbol *node* represents a super-type: variables of type *node* can match against terms with the types in the sub-set:*

$$\mathbb{T}_N := \{ \text{machine, host, hypervisor, machineOS, hostOS, network} \}$$

**Example 1** (Constants and Variables). *We exemplify the declaration of variables  $MA, MB, ZA, ZB$  of types *machine* and *zone* as well as of a constant *bankFrontEnd* of type *machine*.*

<i>bankFrontEnd, MA, MB</i>	<i>: machine</i>
<i>ZA, ZB</i>	<i>: zone</i>



---

Atomic terms can be combined to complex terms, for instance, with function symbols, facts, conditions over terms and facts.

To analyze topologies, we model virtualized infrastructure configurations as graphs. Whereas the basic graph, called *Realization* (cf. Chapter 3), is a unification of vendor-specific elements into abstract nodes, we introduce further graph types to model information flow and dependencies.

**Definition 12** (Graph Types). *A graph type  $G \in \{\text{real}, \text{info}, \text{depend}, \text{net}\}$  is a constant identifier for a type of a graph model:*

- *real denotes a realization graph unification of resources and connections thereof.*
- *info denotes a realization graph augmented with colorings modeling topology information flow.*
- *depend denotes a realization graph augmented with colorings modeling sufficient connections to fulfill a resource's dependencies.*
- *net denotes a realization graph augmented with colorings modeling only the network topology information flow.*

---

#### 4.4.2 Function Symbols and Dependent Terms

---

In order to express properties of graphs and sets we introduce a number of functions, such as a predicate if two nodes are connected in a graph.

**Definition 13** (Function Symbols).  *$\Sigma$  contains a finite set of fixed function symbols.*

- *contains( $S, E$ ) denotes a untyped set membership relationship of a set  $S$  and element  $E$ .*
- *matches( $I, R$ ) denotes the correspondence between an element of the ideal world  $I$  and the real world  $R$ . Both elements  $I$  and  $R$  must have the same type.*
- *edge( $[G := \text{real}]; A, B$ ) is a predicate, which denotes the existence of a single edge between  $A$  and  $B$  with respect to an (optional) graph type  $G$ .*
- *connected( $[G := \text{real}]; A, B$ ) is a predicate, denotes existence of a path between  $A$  and  $B$ , respect to an (optional) graph type  $G$ .*
- *paths( $[G := \text{real}]; A, B$ ) denotes the complete search of all paths between  $A$  and  $B$ , with respect to an optional graph type  $G$ . The resulting type of the function is a set of edge pair sets.*

*The notation  $[A := v]$  denotes an optional argument  $A$  with default constant value  $v$ . We may specify the graph type as a first optional argument delimited by a semi-colon.*

Observe that the graph functions allow an optional graph type argument  $G$  (Definition 12), which specifies the graph type the function is applied to.

We introduce the notion of *dependent terms* to model access to resource properties, such as IP address  $\text{ipadr}(M)$  or image type  $\text{imagetype}(M)$  of a machine  $M$ .

**Definition 14** (Dependent Term Function Symbols). *A dependent term is a function symbol denoting the mapping of constant values to atomic terms.  $\Sigma$  includes a fixed set of constant symbols for dependent terms.*

---

### 4.4.3 Facts, State and Conditions

---

*VALID* aims at reasoning over secure and insecure states of a cloud topology, which we model as a set of known facts.

**Definition 15** (Facts and State). *A Fact represents a piece of knowledge. A state is a set of ground facts (i.e., variable free facts). We express such sets by a dot-operator (“.”), that is, a commutative, associative, idempotent operator, which joins all facts of a state.*

**Example 2** (Facts and State). *We exemplify a state constituted by two facts. The first fact models that a set  $za$  contains the machine  $ma$  as element. The second fact models that  $ma$  and  $mb$  are connected by a path.*

```
contains (za, ma) . connected (ma, mb)
```

**Definition 16** (Condition). *A condition is a conjunction of equalities and inequalities on terms. We define the condition function symbols for equality  $\text{equal}(\text{Term}, \text{Term})$  and less-or-equal  $\text{leq}(\text{Term}, \text{Term})$  over terms as well as negation  $\text{not}(\text{Condition})$  and conjunction operator “& Condition” over conditions with their natural semantics.*

---

### 4.4.4 Goals

---

We define *goals* by specifying an abstract state which constitutes attaining the goal. For an analysis we match a Fact set modeling the goals constrained by a conditions list against the actual analysis state.

**Definition 17** (Goal). *A goal state is a set of positive and negative facts constrained by a (potentially empty) condition list. It is specified with a unique identifier, an optional graph type  $G$  and a variable list as interface. It has the form:*

$$\text{goal } \text{Identifier } ([G := \text{real}]; \text{VariableList}) := \\ \text{PF.NF } C$$

where  $PF$  and  $NF$  are positive and negative fact sets and  $C$  a condition list. The graph type  $G$  determines the graph type of unparametrized graph functions used in the goal.

In principle, it is possible to formulate ideal state goals and attack state goals. The former method often requires an exhaustive state search to report that the goal was reached, whereas attack states can be determined efficiently. In this paper, we focus on attack state specification in Section 4.5.

**Example 3** (Goal). *Let us consider a simple isolation breach attack state, which matches against a state, in which disjoint zones  $ZA$  and  $ZB$  contain machines  $MA$  and  $MB$  respectively, and in which there exists an information flow path between these two machines. It is determined as information flow goal by the graph type  $\text{info}$ . Observe that the goal is defined over variables and can match against any state with constant zones and machines fulfilling this relation and that the matching values must be different.*

```
goal isolation_breach (info; ZA, ZB, MA, MB) := \\ contains (ZA, MA) . contains (ZB, MB) . \\ connected (MA, MB)
```

---

#### 4.4.5 Structured Specifications

---

Specification of our language consist of distinct sections: The *TypesSection* introduces all atomic terms, which will be used throughout the analysis, with their designated types. The type section may have two subsections for real and ideal type declarations. The *InitsSection* specifies initial knowledge on entities. For instance, here one would specify properties of machines that can be used for identifying the machine, such as the machine's IP address as Condition over machine properties. Knowledge specified here can be about ideal and real entities. Further it specifies the knowledge on the structure of the virtualized infrastructure. For instance, it specifies which machine elements are associated with which isolation zones. Note that the topology specified in this section is particularly important to model the system's ideal state. Finally, the *GoalsSection* defines attack and assurance states which are matched against analysis results.

---

#### 4.4.6 Dual Type System

---

One goal of our language is to express comparisons of a desired state and an actual state discovered in a configuration. Sometimes it is necessary to make statements about ideal elements as well as real elements in the same policy statement. For this reason, we introduce the declaration of ideal and real types, that is a *dual type system*.

**Definition 18** (Dual Types). *For each constant or variable symbol, we explicitly declare symbol to be either universal or restricted to the ideal or real model. A declaration in the top-level of the TypesSection means universal, a declaration in the subsections idealTypes and realTypes restricts the declaration to the respective model. The matches( $\cdot, \cdot$ ) fact denotes that two symbols of ideal and real world have a correspondence with each other.*

---

### 4.5 Definition and Specification of Attack States

---

We model the security goals from Section 4.2 as abstract attack states. In case the state is reached, a tool will alert that the corresponding goal has been breached. This approach aims at security analysis by, for instance, model checking.

To facilitate an actual security analysis, one complements these abstract goals with specifications of the ideal state of the system in two areas: *First*, one defines the initial knowledge on entities, that is, properties modeled as dependent terms, such as IP address. *Second*, one defines the knowledge of the ideal structure of the topology as initial state, that is, facts known on contains, matches or edge relations.

---

#### 4.5.1 Operational Correctness

---

For the operational correctness from Section 4.2.1, we model deployment breach as exemplary attack state.

##### **Deployment Breach**

Deployment breach considers in how far VMs are placed on an incorrect hypervisor or physical machine.

**Definition 19** (Deployment Breach). *A deployment breach is an attack state over some virtual machine  $M$  and two different hosts ( $HA, HB$ ), in which  $\text{edge}(HA, M)$ , i.e.,  $M$  is hosted on  $HA$ , is a specified fact, but  $\text{edge}(HB, M)$  was observed.*

```

section types:
  M                : machine
  subsection idealTypes:
    HA              : host
  subsection realTypes:
    HB              : host

section goals:
goal deploymentBreach (real; HA,HB,M) :=
  not( matches(HA, HB) ) . edge(HA,M) . edge(HB,M)

```

After declaring that HA does not match HB, the left-side of the statement contains the matched facts of the ideal world, that is,  $\text{edge}(\text{HA}, \text{M})$ , the right side of the statement the observed fact of the real world  $\text{edge}(\text{HB}, \text{M})$ .

### Unreachability

Unreachability is an attack state that there does not exist a path between a machine and a dependent resource in the dependency graph.

**Definition 20** (Unreachability). *A unreachability is an attack state over some machine M and a resource set  $\{\text{RA}, \dots, \text{RN}\}$ , on which M depends. The attack state is triggered if no dependency path between M and at least one of the needed resources Rl exists.*

---

## 4.5.2 Failure Resilience

---

### Single Point of Failure

Single Point of Failure (SPoF) describes a state in which a virtualized infrastructure critically depends on any single resource. This may be, for instance, that all functionality of the same type are hosted on a single VM host (SPoF: VM host/physical machine), that a machine has only one path through a single router to the network (SPoF: router), or that a that data is stored only once w/o redundancy (SPoF: storage). Formally, we specify a single point of failure as a path goal, that is, that there must not exist only a single path to a critical resource.

**Definition 21** (Single Point of Failure). *A single point of failure is an attack state over any machine M and any two different resources (RA, RB) with equivalent function. A single point of failure exists if only  $\text{path}(\text{M}, \text{RA})$  holds, but  $\text{not}(\text{path}(\text{M}, \text{RB}))$  for any RB.*

In general, a single point of failure exists if there is only one dependency path between a resource and its dependencies. This requires knowledge what the dependencies of a certain resource (type) are and which other resources can fulfill the same function. For instance, for a network single point of failure, one may consider all network switches that connect to the Internet, independently from the ones connecting to the Intranet. We therefore define different attack state goals for different resource types and model the goals with functional classes of resources fulfilling the same purpose.

```

section types:
  M                : machine
  NA, NB           : network
  C                : class

section goals:
goal singlePoF_Net (depend; NA,NB,M,C) :=
  contains(C,NA) . contains(C,NB) . connected(M,NA) .
  not(connected(M,NB))

```

---

## Interdependent Failure Behavior

Fault-tolerant computing requires *independent failure behavior* of all involved agents. For virtualized infrastructures or clouds, this means that mostly that machines from the same class functional must behave independently in face of fail-stop faults and byzantine faults or compromise. Independent failure behavior is a very requirement for replication, most likely for high-resilience environments and critical infrastructures or fault-tolerant configurations.

In practice, independent failure behavior means that any part of the machines stack (including the machines functionality itself) must be implemented independently, more specifically, the machines must be implemented independently, hosted on a different VM operating system, on hypervisors of different type, on different VM host operating systems, on different types of physical machines. This requires a complex logic of functional classes of all levels of the virtualized infrastructure. We call the corresponding attack class *interdependent failure behavior*.

**Definition 22** (Interdependent Failure Behavior). *Interdependent failure behavior is an attack state over two different machines (MA, MB) with the same functional class C and k pairs of resource and associated class, i.e., a specific implementation, such as:*

$$(\{RA1, \dots, RAN\}, CRA), \dots, (\{RK1, \dots, RKN\}, CRK)$$

*We have an attack if for any two machines (MA, MB) of class C, there exists a resource of the same class they both have in their stack.*

---

### 4.5.3 Isolation

#### Zoning & Isolation Breach

We specify an isolation analysis over machines and zones. Machines can be recognized by their properties, for instance an IP or MAC address. By the contains rule, we express that a machine is associated with zone (i.e., that the zone contains the machine). We define a isolation goal by a zoning breach and an isolation breach attack state as follows.

**Definition 23** (Zoning Breach). *A zoning breach is an attack state over a pair of machines (MA, MB) and zones (ZA, ZB), where either MA is declared to be in ZA and not present, or MB is declared not to be in ZB, but was found there in the real state.*

```
section types:
  MA, MB           : machine
  subsection idealTypes:
    ZA, ZB         : zone
  subsection realTypes:
    ZA0, ZB0       : zone

section goals:
goal zoningBreach_Missing (info; ZA,ZA0,MA) :=
  matches(ZA,ZA0).contains(ZA,MA).
  not(contains(ZA0,MA))
goal zoningBreach_Unknown (info; ZB,ZB0,MB) :=
  matches(ZB,ZB0).not(contains(ZB,MB)).
  contains(ZB0,MB)
```

*Isolation breach* is more complex as it incorporates the existence of information flow paths between zones.

**Definition 24** (Isolation Breach). *An isolation breach is an attack state over any pair-wise different variable machines (MA, MB) and zones (ZA, ZB), MA in ZA and MB in ZB, in which there exists a path between MA and MB.*

```

section types:
  MA, MB          : machine
  ZA, ZB          : zone

section goals:
goal isolationBreach (info; ZA,ZB,MA,MB) :=
  contains(ZA,MA) . contains(ZB,MB) .
  connected(MA,MB)

```

This is a straight-forward formulation of Definition 24; it simply says: “Match against any state with the fact that some machine  $MA$  is in some zone  $ZA$  and the fact that some machine  $MB$  is in some zone  $ZB$ . Deduce an `isolationBreach` on the condition that there exists any information flow path between  $MA$  and  $MB$ .”

### Guardian Circumvention

Guardian circumvention is an attack state corresponding to Guardian Mediation from Section 4.2. It means that there exist paths between machines that are not controlled by a trusted guardian.

**Definition 25** (Guardian Circumvention). *Guardian circumvention is an attack state over any pair-wise different variable machines  $(MA, MB)$ , guardian  $G$  and zones  $(ZA, ZB)$ ,  $MA$  in  $ZA$  and  $MB$  in  $ZB$ , in which there exists a path between  $MA$  and  $MB$ , which does not contain the guardian  $G$ . The attack state naturally extends to a set of multiple guardians.*

```

section types:
  G          : guardian
  MA, MB     : machine
  ZA, ZB     : zone
  N          : node
  P          : set

section goals:
goal isolationBreach (info; ZA,ZB,MA,MB) :=
  contains(ZA,MA) . contains(ZB,MB) . connected(MA,MB) .
  contains(paths(MA,MB),P) . not(contains(P, (G,N)))

```

---

## 4.6 Review of Language Requirements

---

We review how *VALID* fulfills the language requirements of Section 4.3.

- *Formal Foundations*: *VALID* is based on the *Intermediate Format* (IF) [AVI03], which is a formal language for set rewriting and state expressions. IF has been used for the specification and automated analysis of security protocols.
- *Expressiveness*: Our language needs to provide three expression layers to reason about resource properties as well as to express set and graph operations. *VALID* provides *dependent terms* for resource properties, such as `ipadr` for an IP address of a VM. Further, *VALID* introduces functions for set membership (`contains`) and graph properties (`edge`, `connected`, and `paths`).

We evaluated *VALID* by modeling the virtualized infrastructure security goals (cf. Section 4.2) as attack states in our policy language. The attack states are expressed based on the existing goal state expressions of IF as well as using the newly defined *VALID* terms and functions. The goals cover

---

operational and security properties, and require to reason over different parts of the infrastructure, such as resource properties, topology, or information flow. In a case study we validated the scope of our security goals. Thus, *VALID* shows to be expressible enough to cover a variety of properties on the security, topology, and operational properties of a virtualized infrastructure.

- *Tool and Standard Independence:* Although *VALID* is based on the Intermediate Format, multiple tools exist that can consume this format, such as, OFMC [BMV05b], CL-Atse [Tur06], and SAT-MC [AC08]. Therefore, *VALID* is not tied to a single vendor or tool. Furthermore, the expressions over states without transitions that are modeled in IF can be translated into other formalisms, such as first order logic, for which a larger set of tools exist (cf. Section 5.4.3).
- *Desired State Comparison:* In *VALID* we added the concept of a dual type system and the term *matches* to reason about elements from the ideal and real infrastructure in the same policy goal. We demonstrated this functionality as part of the security attack state *Zoning Breach* and the operational attack state *Deployment Breach*. Although the existing tools do not have the concept of the dual type system and the *matches* term, an intermediate compiler can translate those to nested types and an explicit attribute matching.

---

## 4.7 Summary

---

We studied virtualized systems security goals in the categories operational correctness, failure resilience, and isolation. We proposed a formal language to express such high-level security goals, which, unlike previous work, covers topological aspects rather than just individual virtual machines. We chose the *Intermediate Format* (IF) as formal foundation of our language because of its support by existing general-purpose model checkers and theorem provers. We demonstrated the ability of our language to efficiently express a diverse set of virtualized systems security goals by giving concrete specifications for a subset of the studied goals.

The policy language is part of a larger virtualization assurance system that combines the extraction of the infrastructure's configuration with the verification of policies, specified in *VALID*, against the infrastructure state. The system integrates the configuration extraction and information flow analysis of Chapter 3 with an automated analysis and policy verification of Chapter 5.





---

## 5 Automated Verification of Security Policies

We present in this chapter a platform for the automated analysis of virtualized infrastructures with regard to given security policies. The platform connects declarative and expressive description languages with state-of-the-art verification methods. The languages integrate homogeneously descriptions of virtualized infrastructures, their transformations, their desired security goals, and evaluation strategies. The employed verification tools range from model checking to theorem proving; this allows us to exploit the different strengths of the methods, and also to understand how to best represent the analysis problems in different contexts.

We differentiate between multiple analysis cases. First, we consider the *static* case where the topology of the virtualized infrastructure is fixed and demonstrate that our platform allows for the analysis of the infrastructure with regard to security policies given in *VALID* (cf. Chapter 4). Even though tools that are specialized to checking particular properties perform better than our generic approach, we show with a real-world case study that our approach is practically feasible. We finally consider also the dynamic case where an attacker can actively change the topology, e.g., by migrating virtual machines. The combination of a complex topology and changes to it by an attacker is a problem that lies beyond the scope of previous analysis tools and to which we can give first positive verification results.

---

### 5.1 Introduction

---

Virtualized infrastructures and clouds form complex and rapidly evolving environments that can be impacted by a variety of security problems. Manual configuration as well as security analysis often capitulate in face of these ever-changing complex systems. The need for automated security assurance analysis is immediate. Given the volatility of virtualized infrastructure configurations as well as the diversity of desired security goals, specialized analysis tools—even though they may have performance advantages—have limited benefits.

As a general approach, we propose to first specify abstract security goals as *desired state* for a virtualized infrastructure in a formal language. For instance, goals can be in the areas that we defined in Chapter 4: *operational correctness* (e.g., “Are all VMs deployed on their intended clusters?”), *failure resilience* (e.g., “Does the infrastructure provide enough redundancy for critical components?”) or *isolation* (e.g., “Are VMs of different security zones isolated from each other?”). Second, we employ a generic analysis tool to evaluate the *actual state*, i.e., the virtualized infrastructure configuration and topology, against this desired state. Thus, we obtain an automated analysis mechanism that can check the infrastructure—and infrastructure changes—against a high-level security policy.

Such an automated analysis can cover two scopes: in the *static* case, we analyze a single state of a virtualized infrastructure against the desired properties. In the *dynamic* case, we represent the actual configuration as a start state and have transitions that can change this configuration. In our example, we consider particularly changes that an intruder can make to the system (within the limits of his access rights), e.g., by migrating VMs to other security zones. The question is whether we can reach an attack state in this way, i.e., a current configuration of the system that violates the required security properties. The dynamic case is an extension to the static case and requires tools that support state transitions.

From engagements with customers running large-scale virtualized infrastructures, we learned that they are interested in a broad range of security goals. Specialized tools can be applied to a subset of these security goals, as we already demonstrated in Chapter 3 for security zone isolation. However, a general approach is desired that can cover this broad range of security requirements. Our goal is to establish general-purpose verification methods as an automated tool for security assurance of virtualized infrastructures. We present

---

a platform that connects declarative and expressive description languages with state-of-the-art verification methods. With such a platform, we can build upon and benefit from the variety of existing tools and methods with their optimizations.

As desired state specification, we take security assurance goals in our formal language *VALID* (cf. Chapter 4) as inputs. As actual state, we lift the configuration of a heterogeneous virtualized infrastructure to a unified graph model. For this, we employ our tool from Chapter 3, which also computes graph coloring overlays, that model, e.g., information flow. We develop a translator that connects these descriptions with the various state-of-the-art verification tools. The translation involves adapting the verification problem to the domain of the respective tool, and property-preserving simplifications and abstractions to support the verification. In particular, the translation does not add false positives or false negatives to the model.

We demonstrate that automated analysis and model-checking of virtualized infrastructures is in general possible, and we exemplify our approach by studying three examples: *zone isolation*, *secure migration*, and *absence of single point of failure* on the network level. The first example is a static case, which determines whether machines from different security zones are connected in an information flow graph. The relevancy of this case was confirmed in a case study with a financial institution. The second example is of dynamic nature, and checks whether an attacker with rights to migrate VMs can reach an attack state, either by migrating a virtual machine through an insecure network (thereby modifying the machine's content) or to a physical host he controls. Secure migration as an example is used to show our first result in verifying dynamic problems. The last example considers resource dependencies and network redundancies in the system, in order to determine the absence of single point of failures. This example belongs to the static case, however requires a different formalization and tools compared to the other static case examples and in fact uses methods from the dynamic case examples.

---

### 5.1.1 Contributions

---

We are the first to apply general-purpose model-checking for the analysis of general security properties of virtualized infrastructures. We propose the first analysis machinery that can check the actual state of arbitrary heterogeneous infrastructure clouds against abstract security goals specified in a formal language. Our approach covers static analysis as well as dynamic analysis and uses a versatile portfolio of problem solver back-ends to benefit from their different methods.

We believe that our experiments with different modeling and analysis strategies (Horn clauses, transition rules) are of independent interest, because the problem instances for security assurance of virtualized infrastructures are structured differently than traditional application domains of model checkers, notably security protocols. In addition, we gained insights on the complexity relations of different problem classes. As a case study, we successfully analyzed a sizable production infrastructure of a global financial institution against the zone isolation goal. We have previously analyzed this infrastructure extensively with specialized tools and found the same problems with this generic approach. We report that our different optimizations allowed us to improve the performance by several orders of magnitude: whereas the non-optimized problem instances did not terminate within several hours, the optimized problem instances completed the analysis in the order of seconds.

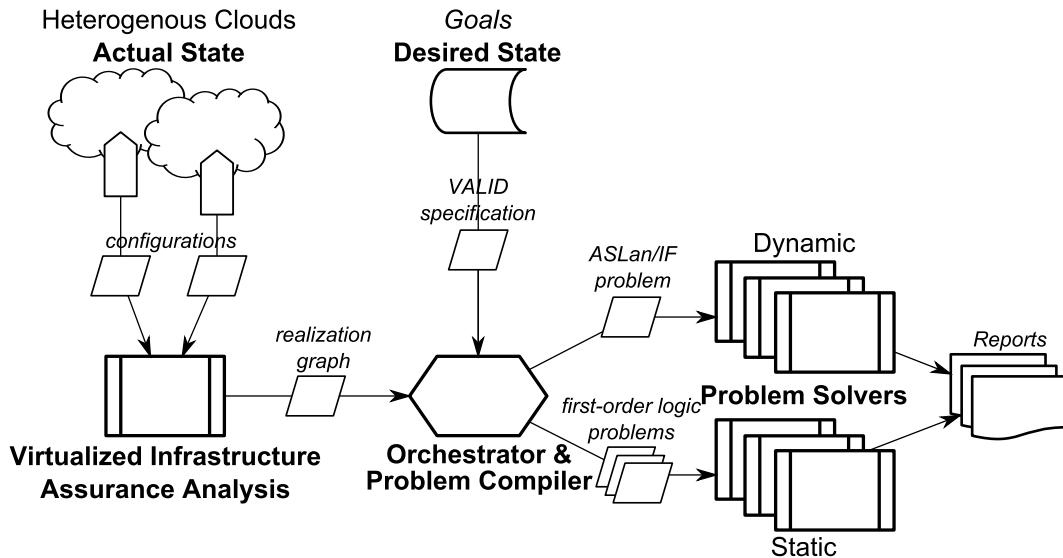
---

### 5.1.2 Architecture

---

We aim at the evaluation of an actual state against a desired state, for which we employ a tool architecture illustrated in Figure 5.1. To specify a *desired state*, we formulate general security goals in *VALID*, our formal language for specifying security goals of virtualized infrastructures (cf. Chapter 4).

To obtain the *actual state* of a virtualized infrastructure, we employ our configuration extraction tool from Chapter 3. It comes with discovery probes for heterogeneous clouds such as VMware, Xen, PowerVM, etc. and extracts their proprietary configuration data as inputs. It lifts the configuration data to a unified graph



**Figure 5.1.:** Architecture for model checking of general security properties of virtualized infrastructures.

representation of the virtualized infrastructure (the *realization model*) and computes the information flow. The tool outputs its graph representations as the actual state basis of our analysis.

We use and compare several state-of-the-art tools for automated verification. The first is the AVANTSSAR platform<sup>1</sup> which consists of three verification backends: OFMC [BMV05b], CL-Atse [Tur06], and SAT-MC [AC08], and all have the common input language ASLan (AVANTSSAR Specification Language). We have focused here on OFMC and made initial experiments with the other two, but due to lack of source code availability and lack of support of Horn clauses in current SAT-MC, we could not run CL-AtSe and SAT-MC on the large scale case study through their web-interface. The particular strength of AVANTSSAR is that we can model a dynamic system with state transitions and check whether a property holds in all reachable states of the system. For the simpler case of analyzing a static infrastructure, a broader range of tools is applicable as we can express verification as deducibility problems in first-order logic. We consider here the automated first-order theorem prover SPASS [WDF<sup>+</sup>09] and the protocol verifier ProVerif [Bla01]. We also made initial experiments with the SuccinctSolver [NNS02]. In general, we expect that tools based on different methods can have complementary strengths and we can benefit from their optimizations.

Our small-scale experiments are carried out with the OFMC model checker. The experiments consist of a static case for zone isolation, and two dynamic cases for secure migration and single point of failure, where we leverage the state transition system to model the dynamic behavior and to formalize path quantification for the single point of failure case. For a case-study experiment we evaluate OFMC, SPASS, and ProVerif.

As the key component for the actual/desired state analysis, we develop an *orchestrator* that takes the graph representation of the actual state and the desired state specification in *VALID* as inputs and compiles problem instances for the solver back-ends. It refines the graph representation, e.g., by abstracting from nodes that cannot affect the analysis goal, compiles the graph to facts, and enhances this problem instance with an analysis strategy and the desired state goals. Note that the AVANTSSAR tools accept *VALID* goals directly as they consume Intermediate Format/ASLan, which is the foundation of *VALID*, and we translate the goals for the other tools.

<sup>1</sup> <http://www.avantssar.eu/>

---

## 5.2 Language Preliminaries

---

Our formal language *VALID* is based on the AVISPA Intermediate Format (IF) [AVI03] language. We will introduce in this section the specification of rules that allow us to specify state transitions for our dynamic cases. Further, we introduce ASLan, the AVANTSSAR Specification Language [AVA10], which is an extension of IF. One of the key extensions of ASLan is the integration of Horn-clauses that allow for complex evaluations within every state of the transition system. This enables us to model different analysis strategies as well as to model access control rules.

---

### 5.2.1 Rules and Goals

---

An IF/ASLan specification consists of an *initial state*, a set of *rules* that give rise to a transition relation, and a set of *goals* that describe a set of states, usually the violations of the security properties. The security analysis shall then determine whether a goal state is reachable from the initial state by applying the rules and perform the state transitions. Moreover, one may add Horn clauses to specify immediate consequences within a single state which we discuss in more detail below in Section 5.2.2.

The rules have the form  $PF.NF.C \Rightarrow RF$  where  $PF$  and  $RF$  are sets of facts,  $NF$  is a set of negative facts (denoted using the operator  $\text{not}(\cdot)$ ), and  $C$  is a set of inequalities on terms. The variables of  $RF$  must be a subset of the variables of  $PF$ . Such a rule is interpreted as follows: we can make a transition from state  $S$  to state  $S'$  if  $S$  contains a match for all “positive” facts of  $PF$ , does not contain any instance that can match a negative fact of  $RF$ , and the inequalities of  $C$  do hold under the given match. The successor state is obtained by removing the matched positive facts of  $PF$  and adding the  $RF$  under the matching substitution.

For example, the following rule expresses that, if an intruder resides at a node  $N$  and there is an edge from  $N$  to another node  $M$ , which is not contained in a particular zone  $z$ , then the intruder can move to  $M$ :

$$\begin{aligned} & \text{intruderAt}(N).\text{edge}(N, M).\text{not}(\text{contains}(z, M)) \\ & \Rightarrow \text{edge}(N, M).\text{intruderAt}(M) \end{aligned}$$

Upon this transition, the fact  $\text{intruderAt}(N)$  is deleted, because it is not repeated on the right-hand side, and the fact  $\text{edge}(N, M)$  remains in the graph since it is repeated on the right-hand side.

The security goals, which we already introduced in Chapter 4, are quite similar to rules in that they have the form  $PF.NF.C$ , i.e., a rule without the right-hand side, and by the same semantics as rules characterize a set of states, usually attack states for state-based safety properties.

---

### 5.2.2 Horn Clauses

---

ASLan introduced the specification of Horn clauses to the transition system to allow for specifying immediate consequences within a state. A Horn clause contains at most one positive fact. A rule is a disjunction of negative facts with one positive facts, which can be written as an implication of the conjunction of only positive facts, e.g.,  $a \wedge b \wedge c \rightarrow x$  or written in ASLan as  $x :- a.b.c$ .

One of the main application is the formalization of access control policies: access rights can be expressed as a direct consequence of other facts that express for instance that an employee is a member of particular group. Horn clauses and the state transition system can mutually interact. First, a transition can change the facts that currently hold, e.g., an employee changes to another group, which has immediate consequences for the access rights via the Horn clauses. Second, the fact representing the (current) access decision can be the condition of another transition rule, for example, where an employee requests access to a resource.

---

In our context, we can also use the Horn clauses to formalize properties of the current graph. E.g., to formalize that `connected()` is the *symmetric* transitive closure of the `edge()` predicate we can specify:

$$\begin{aligned} \text{edge}(B,A) &: - \text{edge}(A,B) \\ \text{connected}(A,B) &: - \text{edge}(A,B) \\ \text{connected}(A,C) &: - \text{edge}(A,B).\text{connected}(B,C) \end{aligned}$$

Introducing or removing edges upon transitions would automatically change the `connected()` relation.

---

### 5.3 Problem Classes

---

During our analysis, we found that the analysis goals for virtualized infrastructures can be structured into orthogonal problem classes, and that different problem classes exhibit complexity tendencies for the solver-backends. We consider problem classes with respect to three criteria on attack states and intruder rules: locality, positivity, and dynamics.

---

#### 5.3.1 Local vs. Global

---

**Definition 26** (Locality). *We call an attack state local if it only exhibits state facts that will be part of the initial state, e.g., `edge()` and `contains()`. We call an attack state global if it exhibits state facts that must be derived by an evaluation over the topology (e.g., `connected()`). We use these terms for the corresponding problem instances, as well.*

Secure migration—in the sense that the intruder cannot reach a state in which he controls the physical host to which a VM was migrated—is a local problem, because the attack state will be formulated on the `edge()` statement between these components. Zone isolation mentioned in the introduction is an example of a global problem, because it needs to consider the connections through-out the infrastructure topology. We expect that local problems can be consistently checked more efficiently than global problems. We also conjecture a positive performance correlation between Horn clause based models and problem solvers with local problems, and between transition based models and problem solvers with global problems.

---

#### 5.3.2 Positive vs. Negative Attack States

---

Attack states formulated in *VALID* can contain positive as well as negative facts.

**Definition 27** (Positivity). *We call an attack state positive if it exclusively contains positive state facts. We call an attack state negative if it contains at least one negative state fact. We use these terms for the corresponding problem instances, as well.*

The secure migration and zone isolation examples are positive problems. A negative attack state is, for instance, the guardian mediation introduced in Chapter 4, which is fulfilled if there exists any connection between a machine and a network that is *not* mediated by a guardian, such as a firewall.

All other factors equal, we conjecture that positive problems can be checked more efficiently than negative problems. Intuitively, for positive attack states it is sufficient to find one fact/variable assignment that matches the attack state, whereas for negative attack states all possible fact/variable assignments for the given statement must be evaluated to conclude that no matching assignment exists.

---

### 5.3.3 Static vs. Dynamic

---

We consider problems that are statically checking whether the actual state fulfills a desired state. By introducing additional transition rules we can allow the intruder to transform the virtualized infrastructure to reach an attack state, and therefore introduce dynamics.

**Definition 28** (Dynamics). *We call a problem instance static if its transition rules and Horn clauses only include topology traversal over the initial state. We call a problem instance dynamic if it contains transition rules or Horn clauses that model intruder capabilities to change the initial state.*

Many example problems presented in Chapter 4 (machine placement, zone isolation, guardian mediation) are static in first instantiation. As soon as we extend the intruder rules/clauses with rights to, e.g., start, stop or migrate machines or to reconnect networks/storage, we obtain dynamic problems. Secure migration introduced above is a dynamic problem.

We expect that static problems can be checked more efficiently than dynamic problems. In the static case, it is more efficient to check for several attack states than in the dynamic case. First-order logic models and tools will be suited for static problems, whereas transition based models and tools will target dynamic problems.

---

## 5.4 Compiling Problem Instances

---

This section discusses how we compile problem instances for the solver back-ends, thus, explains the compiler which is a key component of the architecture in Section 5.1.2. The compiler receives the following inputs: the realization model or derivatives as a graph representation of the actual state and a *VALID* policy as representation of the desired state.

The success and efficiency of the solver back-ends are largely determined by the initial size of the problem instance, by solution strategies that limit the search space complexity, and by problem formulations that match the solvers' capabilities. Therefore, the compiler must strive for a significant complexity reduction while maintaining generality. Because we target sizable real-world infrastructures, the initial problem size may easily be in the order of tens of thousands of nodes and the compiler's pruning prove crucial.

The compiler works in two phases:

- *Graph Refinement*: Reducing the complexity of the graph and representing it as term algebra facts.
- *Strategy Amendment*: Introducing sensible analysis strategies into the problem instance that match the solver's strengths.

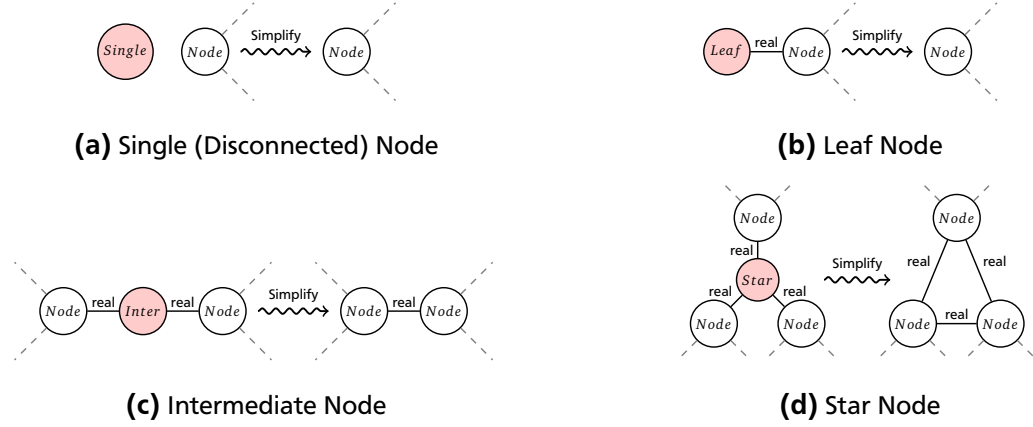
---

### 5.4.1 Graph Encoding and Refinement

---

A (colored) realization model input consists of high-level nodes, such as machine, and low-level nodes, such as `ipInterface`, as well as edges that model the connections between these components. In general, we aim at representing the edges of this graph as `edge()` facts in term algebra and give the problem solvers means to derive graph facts, notably `connected()`.

Real-world virtualized infrastructures consist of tens of thousands low-level components and similarly many edges, an initial complexity that could easily overwhelm the solver back-ends. Therefore, we support the solver back-ends in traversing these graphs efficiently by either abstracting from low-level nodes not impacting the analysis. The graph refinement maintains analysis generality if the pruned node types do neither occur in attack states nor in intruder or topology transformation rules/clauses.



**Figure 5.2.:** Graph Simplifications for Nodes of Different Degrees.

### 5.4.1.1 Simplifying a Graph

Similar to star-mesh transformations employed for the simplification of electrical networks, we apply a simplification algorithm to reduce the size of the realization model graph. Our graph simplification algorithm preserves nodes, which are used in the analysis, and their inter-connectivity. However, it may change the structural properties of the graph and for policies that operate on the graph structure, such as a single point of failure policy with redundant paths, we can only perform a limited graph simplification.

**Definition 29** (Optimization: Graph Simplification). *Given an undirected, vertex-typed graph  $G = (V, E)$  and a set of rules (including goals)  $R$ . We define a graph simplification function  $\text{simplify}$  that takes the current graph  $G$ , the set of rules  $R$ , and a set of candidate nodes  $C(G) \subseteq V$  which may be removed from  $G$ . The function produces  $G' = \text{simplify}(G, R, C(G))$  where for the simplified graph  $G' = (V', E')$  holds  $V' \subseteq V$  and  $|E'| \leq |E|$ .*

*The function is applied recursively on the simplified graph. The recursion terminates and the function returns  $G$  if  $G = G' \vee C(G) = \emptyset$ . The graph simplification has to maintain the invariant that if a rule matches for  $G$  it must also match for  $G'$ , and if a rule does not match for  $G$  it must also not match for  $G'$ . Further the inter-connectivity of non-candidate nodes must be preserved.*

A *candidate* is a node of graph  $G$  that may be removed from the graph in the simplification process. A node is selected to be a candidate based on the following conditions.

- The degree of a candidate node ( $\text{deg}(v)$ ), i.e., the number of adjacent nodes, must be smaller or equal than three. For nodes with higher degrees the simplification algorithm does not reduce the number of nodes and/or edges.<sup>2</sup>
- A node  $v$  must not appear as a constant in a rule's positive, negative facts or term inequalities. Further, the type  $\tau$  of  $v$  must not match or be a sub-type of a variable's type used in a rule.

We differentiate between four cases for a candidate removal based on its degree.

- *Removal of a single (disconnected) node* (Fig. 5.2a): Single disconnected candidate nodes can be simply removed.
- *Removal of a leaf node* (Fig. 5.2b): Candidates that are leaf nodes can be removed as they do not contribute to the connectivity between non-candidate nodes.

<sup>2</sup> Note that the neighbors of a candidate node need to be fully connected after the candidates removal. This leads to a complete graph with  $n = \text{deg}(v)$  nodes and  $e = \frac{n(n-1)}{2}$  edges, where  $e > \text{deg}(v)$  if  $\text{deg}(v) > 3$ , which means an increased number of edges.

- *Removal of an intermediate node* (Fig. 5.2c): Candidates that are intermediate nodes may be removed, but their neighbors have to be connected by an edge, in order to preserve their connectivity. The precondition for removal is that there is no explicit `edge(Node, Node)` fact for the neighbors in any rule. Otherwise a rule would match the edge that would be introduced by connecting the neighbors of the removed intermediate node. This would lead to different rule matching between the origin and simplified graphs, thereby violating our graph simplification invariant. Further, the neighbors must not be connected by an edge already, otherwise our resulting graph becomes a multi-graph.
- *Star-Triangle ( $Y-\Delta$ ) transformation* (Fig. 5.2d): The transformation from a star to a triangle topology does not reduce the number of edges, but decrements the number of nodes by removing the star node. We treat this as three intermediate node removals, i.e., the preconditions for the absence of explicit `edge()` facts have to hold for all the star node's neighbors.

Let us revisit the graph simplification invariants of preserving the rule matching and non-candidate node inter-connectivity, and how our algorithm maintains these invariants.

- *Rule Matching*: Assume we removed a node  $v$  with type  $\tau$ . A rule would not match anymore if a (positive) fact contains  $v$  as a constant or contains a variable that matches or is a super-type of  $\tau$ . However, the candidate conditions prevent the removal of exactly such nodes. The algorithm does not introduce new nodes to the simplified graph, which could alter the rule matching between the origin and simplified graph. Although the algorithm creates new edges, we maintain the precondition that there must be no rule which contains an explicit edge fact that could match the newly connected nodes.
- *Non-Candidate Inter-Connectivity*: When a candidate node is removed we create a complete graph between its neighbor nodes, in order to preserve their connectivity.

We now analyze the termination and complexity of the graph simplification algorithm. The termination conditions are the following:

- *No candidates left*: In each iteration step of the simplification, a subset of candidate nodes are removed from the graph, thereby reducing the set of candidates. The remaining candidates, which have not been removed, trigger the next termination condition. Even though a candidate removal may actually lead to a new candidate in the next round, e.g., due to a degree decrease, the overall set of potential candidates gradually decreases, since we are not adding new nodes to the graph.
- *No simplification performed*: Due to our pre-conditions for candidate removal, e.g., there must be no rule for its neighbors, there may exist candidates that are not actually removed. These candidates thereby do not alter the graph and we terminate the simplification if no more graph changes are performed.

In terms of complexity, we split the algorithm into the following steps:

- *Pre-processing of rules*: We analyze the ASLan/IF rule set and build up a data structure that allows us to query in constant time the existence of a node type or node identifier in the rules.
- *Simplification*: The simplification is dominated by  $|V|$ , i.e., iterating over the vertex set to determine candidates and apply a simplification rule. Each simplification rule is constant, since they only modify a constant and small set of edges for each candidate. The simplification is performed recursively and the number of rounds  $k$  depends on the graph structure. In the worst case, we have a full  $m$ -ary tree, where  $m \geq 4$ , as the input graph and the number of rounds is the depth of the tree. In each round we remove the leaf nodes and thereby decrease the parent node's degree to 1 (a leaf node in the next round).
- *Termination*: In order to determine if any simplifications has been performed, we compare the input and output graphs for equality in Definition 29. In a practical implementation this would be realized with a change flag, thereby the termination condition can be checked in constant time.



---

## 5.4.2 Strategy Amendment for Nodes Connectivity

---

A major part of the solver's strategy will depend on how the graph traversal for determining connectivity is modeled, which we express by symmetric `connected()` facts derived from the `edge()` facts (similar to the Horn clause specification in Section 5.2.2):

$$\begin{aligned} & \text{edge}(A,B).\text{not}(\text{edge}(B,A)) \\ & \Rightarrow \text{edge}(A,B).\text{edge}(B,A) \\ & \text{edge}(A,B).\text{not}(\text{connected}(A,B)) \\ & \Rightarrow \text{edge}(A,B).\text{connected}(A,B) \\ & \text{edge}(A,B).\text{connected}(B,C).\text{not}(\text{connected}(A,C)) \\ & \Rightarrow \text{edge}(A,B).\text{connected}(B,C).\text{connected}(A,C) \end{aligned}$$

Observe that this formulation to compute the `connected()` relation does not change the graph, i.e., `edge()` facts are neither introduced or removed by these rules. While this is a necessity for all evaluations of the graph in the dynamic case, in the static case, we can formalize evaluation procedures that *do* change the graph, for instance rules that remove edges from the graph as soon as they were visited by the evaluation. However this is only applicable to static attack states that argue over `connected()` facts and not `edge()` facts at the same time (such as zone isolation). Our benchmarks show that such changes can improve the performance of our zone isolation example, however only slightly.

We propose an additional translation, which reduces the state complexity significantly. In this case, we imagine an intruder tries to traverse the topology from some start-point and “obtain” nodes he has access to. This avoids the binary fact `connected` and instead uses a unary fact `intruderHas` to represent all members of the largest connected sub-graph that contains the intruder start point.

The transition rules are as follows:

$$\begin{aligned} & \text{intruderHas}(A).\text{edge}(A,B).\text{not}(\text{intruderHas}(B)) \\ & \Rightarrow \text{intruderHas}(A).\text{intruderHas}(B).\text{edge}(A,B) \\ & \text{intruderHas}(A).\text{edge}(B,A).\text{not}(\text{intruderHas}(B)) \\ & \Rightarrow \text{intruderHas}(A).\text{intruderHas}(B).\text{edge}(B,A) \end{aligned}$$

For large graphs, the restriction of analyzing such chunks rather than the full `connected`-relation means substantial savings: roughly speaking, the number of derivable facts is in the worst case linear for the `intruderHas` strategy, while the number of `connected` facts is quadratic. This optimization requires, however, that we have to select one start point for the `intruderHas()` computation and thus get the verification of isolation from other zones only for that selected start point. In case a `connected(A,B)` fact is used in a security goal, we can translate it to `intruderHas(A).intruderHas(B)`.

Depending on the used solver or back-end, the evaluation which nodes the intruder can obtain can either be expressed by a means of transition rules (as above) or in first-order logic using Horn clauses. We showed the specification of the `connected(A,B)` using Horn clauses already in Section 5.2.2. Similarly, the `intruderHas` strategy can be specified with Horn clauses as shown in the following:

$$\begin{aligned} & \text{intruderHas}(B) : - \text{edge}(A,B).\text{intruderHas}(A) \\ & \text{intruderHas}(B) : - \text{edge}(B,A).\text{intruderHas}(A) \end{aligned}$$

In addition to the graph analysis model, we need to introduce intruder rules for the dynamic analysis to model his capabilities to modify the infrastructure. They are highly dependent on the scenario, but can easily be modeled by introducing new facts as well as transition rules or Horn clauses. We exemplify this by modeling the secure migration problem in Section 5.5.

---

### 5.4.3 Encoding Static Problems into FOL

---

In case of static problems, such as zone isolation, we do not need to consider transition systems but can rather encode the problem into “static” formalisms like first-order logic (FOL) and alternation-free least fixed point logic (ALFP) for which mature tools exist. We now show that we can effectively use such tools as an alternative to the model-checking approach in the static case. We study the use of the SuccintSolver for (ALFP) [NNS02], the FOL theorem prover SPASS [WDF<sup>+</sup>09] and the protocol verifier ProVerif [Bla01].

The example of zone isolation can be expressed as an initial set of facts representing the graph structure, a set of Horn clauses expressing the graph traversal as shown in the previous section and in Section 5.2.2, as well as a predicate that an intruder can reach a machine in another different security zone.

The SuccintSolver [NNS02] is an effective tool for computing the least fixedpoint, i.e., all facts that are derivable by the ALFP clauses from the given facts, of an ALFP specification.

The next tool we use is the generic first-order theorem prover SPASS [WDF<sup>+</sup>09] which is based on resolution. A challenge is that we want a model of the symbols where different constants always represent different elements which cannot be enforced directly. In our zone isolation breach we require that VMs of *different* security zones are connected. For the inequality of zones, we need to specify this as axioms for zones.

Concretely, the following listing shows the zone inequality definitions for three zones, the starting point of the intruder traversal at cluster with id `refid1`, `intruderHas` traversal, and the zone isolation policy. We omitted parts of the problem instance files that specify the full graph.

```
...
formula (intruderHas (node (cluster (refid1)))).
formula (not (equal (zone1, zone2))).
formula (not (equal (zone3, zone1))).
formula (not (equal (zone3, zone2))).
formula (forall ([A, B], implies (and (intruderHas (A), edge (A, B)), intruderHas (B)))).
formula (forall ([A, B], implies (and (intruderHas (A), edge (B, A)), intruderHas (B)))).
end_of_list.
list_of_formulae (conjectures).
formula (exists ([ZA, ZB, MA, MB],
  and (contains (zone (ZA), node (vmachine (MA))),
    and (contains (zone (ZB), node (vmachine (MB))),
    and (intruderHas (node (vmachine (MA)))),
    and (intruderHas (node (vmachine (MB)))),
    and (not (equal (ZA, ZB), true)))).
end_of_list.
```

**Listing 5.1:** SPASS input file with `intruderHas` strategy and zone isolation goal.

We finally consider the ProVerif tool [Bla01] which is also based on resolution but dedicated to security problems formulated by Horn clauses, and therefore often faster than SPASS. Like in SPASS, we have to axiomatically introduce here the inequality of zones.

#### Static Problems beyond FOL

Consider the goal of the absence of single point of failures for network links, i.e., that a network contains sufficient redundancies, so that failure of a single node does not disrupt communication.<sup>3</sup> More formally, let us consider a network and the dependability constraint  $depend(n_1, n_2)$  between two nodes  $n_1$  and  $n_2$ . Then we require that there are at least two disjoint paths (using disjoint nodes) in the network from  $n_1$  to  $n_2$ . Even in the static case (when the network topology cannot change) this problem, i.e., paths quantifications, is beyond the expressiveness of first-order logic, because we want to quantify over a set of sets.

---

<sup>3</sup> In Chapter 4 we considered a different form of single point of failure that can be expressed as a goal state: when a node depends on a particular resource, then it is connected to more than one node to provide that resource.

As a consequence, we cannot use the solvers SPASS, ProVerif, and Succinct Solver. Also, the standard approach to specify the security property as a set of *VALID* or ASLan goals (even using Horn clauses to evaluate the graph) is not applicable, because that would also be FOL expressible relations. However, we can specify a transition system in ASLan to express a game that has a solution (expressed as a set of goal states) if and only if there exists no single point of failure. We demonstrate this game for the absence of single point of failure in Section 5.5.3.

---

## 5.5 Model-Checking a Virtualized Infrastructure

---

In this section, we study three example problems, namely zone isolation, secure migration, and absence of single point of failure, and demonstrate how these problems can be analyzed using a model checker. We apply model checking on small infrastructure examples to demonstrate the approach for different problems, and we will analyze a large-scale infrastructure with regard to zone isolation in Section 5.6. We structure this section analogously to the architecture Section 5.1.2 where for each example problem, we first specify the desired state in *VALID* or ASLan goals along with the required language primitives. Second, we introduce the actual state, that is the infrastructure examples we analyze. Third, we discuss specialties of compiling the corresponding problem instance, the problem solvers employed and their output for the analysis.

---

### 5.5.1 Zone Isolation

---

We consider the following scenario to illustrate the zone isolation security goal: an enterprise network consists of three security zones, namely a *high* security zone containing confidential information, a *base* security zone for regular IT infrastructure, and a *test* security zone. Any machine in one zone should not be able to communicate with a machine from a different zone, and network isolation is realized using VLANs.

#### Desired State

To have the solvers check violations of zone isolation, we define an attack state `isolation_breach`, which checks whether any two machines of any two different security zones are connected.

**Definition 30** (Goal: Zone Isolation). *The isolation breach attack state matches if any two disjoint zones ZA and ZB contain machines MA and MB respectively, and in which there exists an information flow path between these two machines denoted by connected. It is determined as information flow goal by the graph type info.*

```
goal isolation_breach(info;ZA,ZB,MA,MB):=
  contains(ZA,MA) . contains(ZB,MB) .
  connected(MA,MB) & not(equal(ZA,ZB))
```

Furthermore, the *VALID* policy requires a specification of the membership of machines to specific zones. For example, `contains(high, vm1)` denotes that *vm1* is part of the *high* security zone.

#### Actual State

SAVE (cf. Chapter 3) discovers the given infrastructure and captures all low-level configuration details and resource associations. SAVE performs an information flow analysis with the different security zones as information sources and produces an information flow graph for the infrastructure.

## Model-Checking

Based on the actual state provided by SAVE, our compiler will generate a representation of the (potentially refined) information flow graph in edge() facts and node constants. Since we are dealing with a static problem, we use the efficient intruderHas modeling for graph traversal, and transform the goal accordingly. The output is ASLan for OFMC and a variety of first-order logic languages used by the static problem solvers.

Suppose the VLAN identifier of a machine *vm2* in the *test* zone was misconfigured and is identical to the VLAN ID of a machine *vm1* from the *high* security zone. OFMC will provide us with such an attack state (reduced for brevity) indicating a zone isolation breach as shown in Listing 5.2.

```
SUMMARY
UNSAFE
PROTOCOL
  zone_isolation.if
GOAL
  isolation_breach

% contains(zone(high),node(machine(vm1)))
% contains(zone(test),node(machine(vm2)))
% intruderHas(node(machine(vm1)),i)
% intruderHas(node(machine(vm2)),i)
```

Listing 5.2: OFMC Output for Zone Isolation Breach

## 5.5.2 Secure Migration

Secure migration is a problem often encountered in practice which was also highlighted by Oberheide et al. [OCJ08]. Secure Migration is an interesting problem as its very nature requires a dynamic modeling. However, we do not claim to solve it completely with this work, as is a complex endeavor in which many factors (network and storage connections, VLAN associations, correct configuration of VMs, machine contracts, etc.) need to be considered. Still, we want to demonstrate the principles of dynamic analysis with a simplified example of this problem class. We leave a full-scale analysis of secure migration of a production system for future work.

We consider the topology depicted in Figure 5.3 for our scenario: five hosts, where HostC is controlled by a malicious administrator, are connected to two networks. The malicious administrator can migrate virtual machines between hosts as indicated by the *migrate* edges. There is one VM running on host *HostA*.

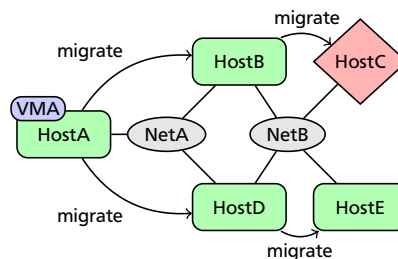


Figure 5.3.: Migration Scenario Topology

## Desired State

We study two exemplary instantiations of the problem of secure migration. The attack state *vm\_breach* asks whether the intruder can migrate a virtual machine from a secure environment to a physical host to which he has root access (in order to perform attacks demonstrated by Rocha et al. [RC11]). The attack

state `insecure_migration` asks whether an intruder can migrate a VM through an insecure network in order to manipulate the VM (cf. attacks demonstrated by Oberheide et al. [OCJ08]).

We define these goals in *VALID*, for which we introduce the unary facts `intruderAccess()` and `root()`, and the binary fact `migrate()`. These model the intruder's access capability set of root access (typically to a given host) and machine migration between two hosts. These facts have the following signature:

$$\begin{aligned} \text{intruderAccess} &: \text{fact} \rightarrow \text{fact} \\ \text{migrate} &: \text{host} * \text{host} \rightarrow \text{fact} \\ \text{root} &: \text{node} \rightarrow \text{fact} \end{aligned}$$

The fact `intruderAccess()` models the set of all access rights the intruder has, that is, it has the semantic that any term enclosed by the fact belongs to the intruder's access capabilities. The fact `root()` models administrator rights on the enclosed node.

We model virtual machine migration in the following way.

**Definition 31** (Migration). *The capability of migrating a VM MA from host HA to HB is expressed as Horn clause `canMig` that incorporates the intruder access to migrate between these two hosts, that both hosts are connected to the same network NA, and host HA is running the VM.*

*Migration is a transition rule that removes the association of a VM MA to a host HA, and adds an association to a new host HB in case fact `canMig` matches.*

$$\begin{aligned} \text{canMig}(MA, HA, HB, NA) &: - \text{edge}(HA, MA). \text{edge}(HA, NA) \\ & \quad . \text{edge}(HB, NA). \text{intruderAccess}(\text{migrate}(HA, HB)) \\ \text{edge}(MA, HA). \text{canMig}(MA, HA, HB) &\Rightarrow \text{edge}(MA, HB) \end{aligned}$$

The goals are defined in *VALID* in the following way:

**Definition 32** (Goal: VM Security). *The VM breach attack state matches if there is a `root()` fact on a host HA in the intruder's access capability set and a VM MA being connected to the host.*

```
goal vm_breach(real; HA, MA) :=
  intruderAccess(root(HA)).edge(MA, HA)
```

**Definition 33** (Goal: Secure Migration). *The attack state for insecure migration is the following. The intruder can migrate a VM MA from host HA to HB, and he has root access to a host HC that is connected to the same network.*

```
goal insecure_migration(net; HA, HB, HC, MA, NA) :=
  canMig(MA, HA, HB).
  intruderAccess(root(HC)).
  edge(HA, NA).edge(HB, NA).edge(HC, NA)
```

If the intruder has root access to a host connected to the migration network, he can mount network attacks, such as ARP spoofing, in order to create a man-in-the-middle attack and intercept the VM migration.

### Actual State

We model the access capabilities of the intruder for our scenario in the following way.

- `intruderAccess(root(hostC))`
- `intruderAccess(migrate(hostA, hostB))`
- `intruderAccess(migrate(hostA, hostD))`
- `intruderAccess(migrate(hostB, hostC))`
- `intruderAccess(migrate(hostD, hostE))`

The network information flow graph for the scenario is generated by SAVE.

---

## Model-Checking

Unlike in the previous static example, we had to explicitly model the dynamic behavior of the intruder, i.e., machine migration, and its effects on the infrastructure. We modeled that as transition rules with restrictions based on access privileges of the intruder. Since we are dealing with a dynamic problem, we have to use a tool from the AVANTSSAR tool chain, for instance we use OFMC.

OFMC found the following attack states (reduced for brevity) for our scenario. First for VM breach where a VM is migrated to an intruder controlled host. As shown in Listing 5.3, OFMC finds this attack state for *vm\_breach* due to the migration of *VMA* to *HostB*, and then to *HostC*. Second, in Listing 5.4 an attack state for *insecure\_migration* is reached by the migration of *VMA* to *HostD*, then to *HostE* and intercepted by *HostC* due to the connection to the same network *NetB*.

```
INPUT
  migration.if
SUMMARY
  ATTACK_FOUND
GOAL: vm_breach

% Reached State :
%
% intruderAccess (root (node (host (hostC))), i)
% edge (node (machine (vma)) . node (host (hostC)), i)
```

**Listing 5.3:** OFMC Output for VM Breach

```
INPUT
  migration.if
SUMMARY
  ATTACK_FOUND
GOAL: insecure_migration

% Reached State :
%
% canMig (node (machine (vma)) . node (host (hostD)) . node (host (hostE)), i)
% intruderAccess (root (node (host (mc))), i)
% edge (node (host (hostD)) . node (network (netB)), i)
% edge (node (host (hostE)) . node (network (netB)), i)
% edge (node (host (hostC)) . node (network (netB)), i)
```

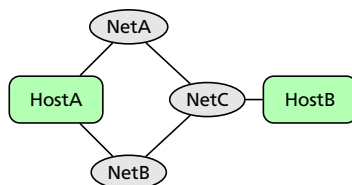
**Listing 5.4:** OFMC Output for Insecure Migration

---

### 5.5.3 Absence of Single Point of Failure

---

We consider the topology illustrated in Figure 5.4 for our scenario to demonstrate the absence of single points of failure for network links. We have two hosts that are depended on each other and connected through a combination of three networks.



**Figure 5.4.:** Single Point of Failure Scenario Topology

## Desired State

The goal of the absence of single point of failure for network links is not expressible in FOL, *VALID* or ASLan goals as discussed in paragraph 5.4.3. Therefore, we construct a game using transitions in ASLan that has a solution if and only if there exists no single point of failure.

This game works as follows for a single dependency constraint  $depend(n_1, n_2)$  (if there are several such constraints, one must start each as a separate game). We have two phases in which sets  $S_1$  and  $S_2$  of nodes are collected. Note that this covers the case for two-way redundancy. In the first phase we start with  $S_1 = \{n_1\}$  and follow edges from a member of  $S_1$  to a non-member that we then add, until we have reached  $n_2$  and start the second phase. We begin similarly with  $S_2 = \{n_1\}$  and follow an edge from a member of  $S_2$  to a node that is not part of *either*  $S_1$  and  $S_2$  that we add to  $S_2$  until we have reached  $n_2$ . Then  $S_1$  and  $S_2$  represent nodes for two disjoint (except for start and end nodes) paths from  $n_1$  to  $n_2$ . Since the transition system allows to choose the edge to follow, the goal state  $n_2 \in S_2$  is reachable if and only if such disjoint paths exist.

In the following are the transition rules modeling this game. The first one starts the first phase, the second one traverses nodes in the first phase, and the third one terminates the first phase and starts the second phase. The fourth rule traverses nodes in the second phase.

$$\begin{aligned} &not(round1).not(round2).depend(A,B) \\ &\Rightarrow round1.depend(A,B).inS1(A) \\ \\ &round1.depend(A,B).inS1(X).edge(X,Y).not(inS1(Y)) \\ &\& not(equal(Y,B)) \\ &\Rightarrow round1.depend(A,B).inS1(X).inS1(Y) \\ \\ &round1.depend(A,B).inS1(X).edge(X,B) \\ &\Rightarrow round2.depend(A,B).inS1(X).inS1(B).inS2(A) \\ \\ &round2.depend(A,B).inS2(X).edge(X,Y).not(inS1(Y)) \\ &.not(inS2(Y)) \& not(equal(Y,B)) \\ &\Rightarrow round2.depend(A,B).inS2(X).inS2(Y) \end{aligned}$$

Here we use special facts *round1* and *round2* to separate the different phases and *inS1* and *inS2* to denote the members of  $S_1$  and  $S_2$ .

The following goal is reached when the second phase terminates, and thereby identified a second disjoint path between A and B.

```
goal spof_absence (A,B,X) :=
  round2.depend(A,B).inS2(X).edge(X,B)
```

Edge symmetry is not handled by the previously shown transitions and the goal, and has to be modeled explicitly with another set of transitions and a goal. For our scenario, we also have to specify the dependency between *HostA* and *HostB* using the *depend* term.

With regard to termination, in both rounds the edge set is reduced by removing the traversed edge, and the vertex sets of visited vertices is expanded on each edge traversal with the unvisited destination vertex. The termination happens when: 1) no edges left to traverse, 2) all reachable nodes have been visited in either *round1* or *round2*, or 3) the destination vertex is B when in *round2*. Effectively we are performing two DFS traversals starting from node A that stop when reaching node B. Whereas the first one triggers the second one upon reaching B and the second one leads to the goal state when reaching B.

---

## Actual State

The network information flow graph for the scenario is generated by SAVE. The graph simplification must not perform star-triangle simplifications as these change the structural properties of the graph. In fact they remove possible single point of failures, i.e., stars, and replace with with a redundant mesh.

## Model-Checking

Since we are dealing with a static problem that cannot be encoded in first-order logic, we modeled this goal in such a way that an attack state is actually a satisfaction of the goal, namely there are no single point of failures. This is contrary to the previous two examples, where an attack state always denoted a breach of a security goal.

For our scenario, the model-checker OFMC will not reach an “attack state”, therefore the infrastructure contains a single point of failure. **Removing the SPoF:** Now we consider connecting *HostB* also to *NetB*, therefore we get a second disjoint path from *HostA* to *HostB*. OFMC produces the output in Listing 5.5 showing the two disjoint paths (reduced to *inS1* and *inS2* facts, and reordered):

```
INPUT
  spof.if
SUMMARY
  ATTACK_FOUND
GOAL: spof_absence

% Reached State :
%
% inS1 (node (host (hostA)) , i)
% inS1 (node (network (netB)) , i)
% inS1 (node (host (hostB)) , i)
% inS2 (node (host (hostA)) , i)
% inS2 (node (network (netA)) , i)
% inS2 (node (network (netC)) , i)
```

**Listing 5.5:** OFMC Output After Removing the SPoF: Absence of Single Point of Failure

---

## 5.6 Case Study for Zone Isolation

In this section, we analyze a real and large-scale production environment of a global financial institution. The infrastructure consists of approximately 1,300 VMs and its realization model modeling all networking and storage resources consists of approximately 25,000 nodes and 30,000 edges. The infrastructure is divided into several security zones, each containing multiple clusters, and models networking up to Layer 2 separation on VLANs and storage providers up to separation on file level. We have already analyzed this virtualized infrastructure extensively with specialized tools and know which attack states to expect. Given the large initial size of the actual state, this case study provides a suitable test environment for the subsequent performance analysis.

Whereas our compiler translates the problem instances to the different static and dynamic problem solvers introduced in Section 5.1.2, we focus the performance evaluation on three tools: SPASS and ProVerif for the static case, and OFMC for both the static and dynamic case. We analyze various optimization and modeling techniques introduced in Section 5.4 to establish their effects in practice. We have also performed initial experiments with SAT-MC and CL-AtSe, but could not apply them to the large case study. The reason is that the tool versions which support ASLan with Horn clauses were only available as third-party hosted web services, which is problematic to use given the sensitive nature of our case study data.

We are focusing in this evaluation on two specific clusters (we call them *Cluster1* and *Cluster2*) and their corresponding information flow graphs, for which we know that *Cluster1* has an isolation problem, where VMs of different security zones are able to directly communicate due to a VLAN configuration error, and *Cluster2* is safe in terms of VM zone isolation.



## Graph Refinement

We first measure the simplification of the information flow graphs for the different clusters in terms of the number of edges and nodes. The information flow graph of *Cluster1* consists of 14386 nodes and 17817 edges. We achieve a reduction of the graph by 13428 nodes and 16860 edges, resulting in a graph with only 958 nodes and 957 edges. The algorithm performs this simplification in 0.18 seconds. *Cluster2* has a smaller information flow graph with 6218 nodes and 7543 edges. The graph reduction completes within 0.06 seconds and results in a graph with 359 nodes and 358 edges.

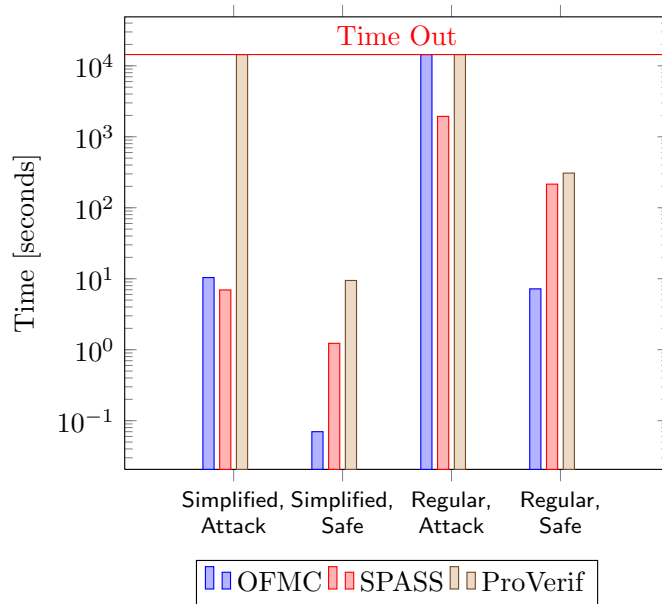
## Zone Isolation

We are now evaluating the analysis of the zone isolation goal for the large-scale infrastructure. For our evaluation, we consider all analysis cases for the following parameters: attack/safe, simplified/non-simplified graph, and different graph traversal models. *Attack* denotes an isolation breach and *Safe* denotes secure isolation.

For the graph traversal modeling using `connected()` in form of Horn clauses or transition rules, all tools we are evaluating either run out of memory (OFMC) or do not terminate within our time limit of 4 hours. We therefore focus our detailed performance analysis on our intruderHas graph traversal model with the following analysis cases.

- *Simplified Graph: Attack 1, Safe 2*
- *Non-Simplified: Attack 3, Safe 4*

The time measurements of the analysis cases for the different tools are depicted in Figure 5.5.



**Figure 5.5.:** Time measurements (on logarithmic scale) for analysis cases of zone isolation.

The measurements show that ProVerif is only able to analyze the *Safe* configuration, because in the other case it does not terminate within our time frame of 4 hours. Since ProVerif is based similarly on resolution as SPASS (which terminates within the time limit for all problems), we suspect that the pre-processing of rules in ProVerif may be the cause.

OFMC yields good performance results and is fast for analyzing such a large-scale infrastructure. We noticed a problem in analyzing the vulnerable cluster with the non-simplified graph, that is OFMC runs out of memory. SPASS terminates for all analysis cases and is faster for case 1 as OFMC.

---

## Discussion

The analysis of a large-scale infrastructure with regard to the zone isolation goal gave us insights into the efficiency of our modeling and the employed problem solvers. We learned that our initial modeling of `connected()` facts using Horn clauses or transitions were only applicable for small infrastructures and not for such real-world scenarios. Therefore, we developed the more efficient modeling of using `intruderHas()` facts for graph traversal, which made the analysis in a reasonable time frame possible. The complexity of this graph traversal is only linear to the number of edges, whereas the graph traversal using `connected()` yields a quadratic complexity.

Furthermore, we learned that problem solvers were overwhelmed by the detailed modeling of the infrastructure in form of our realization model. In case of security goals concerned with graph connectivity, we developed a graph refinement algorithm that simplifies the realization graph, but preserves its connectivity properties. The combination of efficient graph traversal modeling and graph simplification yielded results in the order of seconds for the analysis of our scenario infrastructure.

In terms of employed problem solvers, SPASS and OFMC performed best for our scenario.

---

## 5.7 Summary

---

In this chapter we demonstrated our system architecture and approach for the automated verification of virtualized infrastructures. We are able to specify a variety of security goals in a formal language and validate heterogeneous infrastructure against them. We are the first to employ general-purpose model-checker and theorem provers for this matter.

We studied three examples of static and dynamic problems, namely zone isolation, secure migration, and single point of failure. For each problem, we showed how to specify goals in the formal languages and proposed efficient modeling strategies. We successfully demonstrated the automated verification of these examples against small infrastructures. Finally, we also validated a large-scale infrastructure against the zone isolation secure goal and showed the practical feasibility of our approach.

---

## 6 Dynamic Information Flow Graphs

We introduce a static information flow analysis for dynamic systems. Based on user-configurable trust assumptions, our approach computes an information flow graph on top of a system model graph. The edges in this information flow graph are annotated with dependencies on the trust assumptions' conditions, which operate on node attributes and connectivity. A dynamic system model is described as a graph delta of incremental and decremental node and edge changes as well as node attribute changes. Our differential analysis computes the impact of a system model graph delta on the information flow graph based on the information flow edges' dependencies. We apply our approach to the practical and important problem of tenant isolation in dynamic virtualized infrastructures.

---

### 6.1 Introduction

---

Isolation is a fundamental security requirement in any multi-level security system. The non-interference property [GM82] is a strict formalization of isolation: Inputs and outputs are classified as either *low* or *high*, and a computation on *low* values must not influence *high* outputs and vice versa. Less strict and practical variants of non-interference have been proposed, which allow for instance mediated communication between different security levels using channel control [Rus92]. Alternatively, access control models for multi-level security systems exist, such as, Biba [Bib77] for integrity, Bell-LaPadula [BLP76] for confidentiality, and Chinese Wall [BN89] for confidentiality with conflicting parties.

These fundamental security models find their application in practical systems security. For instance in virtualization, the sHype [SJV<sup>+</sup>05] hypervisor mediates inter-VM communication and enforces the aforementioned access control models. Rueda et al. [RVJ09] analyzes VM access control policies using information flow graphs to verify inter-VM flows. The TVDc [BCP<sup>+</sup>08] approach enforces access control on the entire virtualized infrastructure level and not just on the hypervisor. Similar approaches have been proposed for the Android operating system, such as, domain isolation [BDD<sup>+</sup>11], taint tracking [EGC<sup>+</sup>10], and permission analysis using graph reachability [BDD<sup>+</sup>12].

In general we can classify the isolation approaches as either *static* or *dynamic*. The static approaches operate on a model of the system and compute potential information flows, in order to make a policy decision on illegal flows. The dynamic approaches monitor the running system for actual flows to detect or block illegal ones. A similar classification is done in program analysis where static approaches operate on the source code of the program, and dynamic approaches analyze the executing program. One benefit of the static approach is that we compute all possible flows whereas in the dynamic approach we only see the current actual flows. However, the static approach operates on a model of the system. In dynamic systems we need to ensure that the model is kept in sync with the actual system, otherwise we have delays in the detection of violations or miss transient violations altogether. Previous static approaches, such as [RVJ09] and our approach of Chapter 3, only operate on static system models. In this chapter we pursue a static analysis approach of dynamic systems based on system change events and a differential analysis. We apply our approach to the case study of isolation in dynamic virtualized infrastructures. However our approach is general enough to be applied also in other domains, such as attacker propagation in digital-physical environments or access control configurations.

Our approach works in two phases: the *initial* phase and the *differential* phase. The initial phase takes a current snapshot of the system modeled as a graph. A directed information flow graph is computed as an overlay graph on top of the system model graph. Using the information flow graph we can compute reachability between any two nodes, in order to verify isolation policies. The information flow edges are constructed based on a set of user-defined flow rules that capture trust assumptions on system elements

---

and their isolation properties. The constructed edges are dependent on the flow rules' conditions on node attributes and connectivity. This lays the foundation for the dynamic analysis because we record the existence requirements for each edge and can verify if these requirements still hold in a changing system. In the differential phase, we obtain a system model change as a graph delta, which describes incremental and decremental node and edge changes as well as node attribute changes. We compute the impact of the system model change on the information flow graph based on evaluating the flow rules for new nodes and edges, remove information flow edges for deleted elements, as well as using the edge's dependencies on node attribute or connectivity changes.

## Contributions

In summary we make the following contributions.

- We propose the novel concept of information flow graphs constructed from user-defined flow rules. The flow rules capture trust assumptions on isolation in system components based on their attributes and connectivity. This leads to a generic and user-configurable approach that we apply to the case study of isolation in virtualized infrastructures. We analyze the correctness and complexity of our approach, in particular we adapt a firewall fault model to analyze flow rules sets.
- We establish dynamic information flow graphs that are updated based on system model changes, including incremental, decremental, node property, and resulting connectivity changes. This enables a differential information flow analysis for dynamic systems. We apply our dynamic approach also to the case study of isolation in virtualized infrastructures in combination with a system that provides system model changes.

---

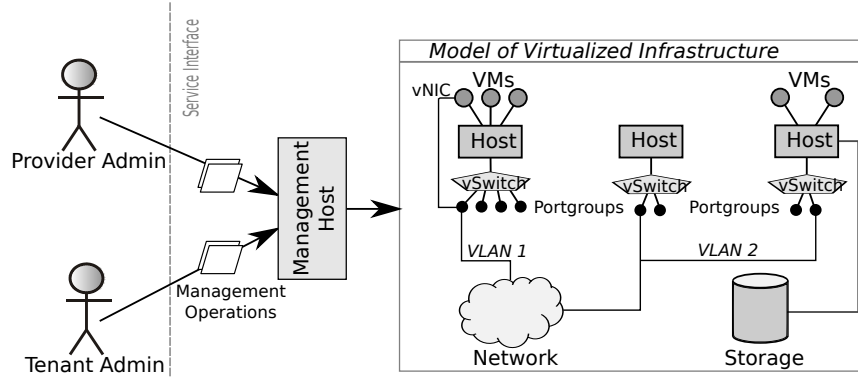
## 6.2 Isolation in Virtualized Infrastructure

---

Multi-tenant virtualized infrastructures offer self-service access to a shared physical infrastructure with compute, network, and storage resources. While administrators of the provider govern the infrastructure as a whole and the tenant administrators operate in partitioned logical resource pools, both groups change the configuration and topology of the infrastructure. For example, they create new machines, modify or delete existing ones, causing large numbers of virtual machines to appear and disappear, which leads to the phenomenon of server sprawl [GR05]. Therefore, self-service administration, dynamic provisioning and elastic scaling lead to a great number of configuration and topology changes, which results in a complex and highly dynamic system.

Misconfigurations and insider attacks are the adverse results of such complex and dynamic systems. Indeed, even if committed unintentionally, misconfigurations are among the most prominent causes for security failures in IT infrastructure [OGP03]. Notably, according to studies by ENISA [ENI09] and CSA [CSA10], operational complexity, which leads to misconfiguration and security failures, as well as isolation failures are among the top threats in virtualized infrastructures. Isolation failures put both the provider as well as the consumers at great risk due to potential loss of reputation and the breach of confidential data. Further, malicious insiders and their attacks are considered a top, very high impact security risk. Consider an example of isolation breach from misconfiguration, which we encountered in the security analysis of a financial institution's in-house VMware-based production cloud: An administrator performed a wrong VLAN ID configuration change leading to an unnoticed network isolation breach between the high-security and the test security zone. The goal of our approach is to compute an information flow analysis in such rapidly changing systems.

In Fig. 6.1 we illustrate our model of a virtualized infrastructure, which consists of (virtualized) computing, networking and storage resources that are configured through a well-defined management interface. In particular, we illustrate the networking part in more detail. Physical hosts and their hypervisors provide networking to VMs by virtual switches that connect the VMs to the network. A virtual switch contains virtual ports, to which the VMs are connected via a virtual network interface card (vNIC). Virtual ports



**Figure 6.1.:** Model of a Virtualized Infrastructure

are aggregated into *port groups*, which apply a common configuration to a group of virtual ports. Virtual LANs (VLANs) allow a logical separation of network traffic between VMs by assigning distinct VLAN IDs to the port groups. Our network model is focused on the OSI Layer2.

From an isolation and trust assumptions point of view, two VLANs are logically isolated from each other if they are configured with distinct VLAN ID values (and not configured to 0). However, if two port groups have the same VLAN IDs, but their underlying virtual switches are without a physical network connection, then they are also physically isolated. On the compute level and an arguable assumption is that hypervisors isolate VMs, i.e., no side channels [RTSS09] exist. It is crucial to allow user configurable/extensible rules that capture those different and arguable assumptions. The goal of our approach is to capture user-dependent trust assumptions in rules that guide our information flow analysis. The output of the analysis is tightly dependent to the conditions of the rules and these conditions may be invalidated due to system model changes, which leads to a complex analysis.

### 6.3 Constructing an Information Flow Graph using Flow Rules

In this section we lay the foundation for the fully dynamic information flow analysis by constructing an overlay information flow graph on a given system model graph using flow rules. We formalize both the system and information flow models, as well as defining the flow rules and their matching. We introduce an algorithm for the first-matching of flow rules and discuss a well-ordering for rules sets. Important for the dynamic analysis is to capture the dependencies of information flow edges on the rules' conditions. Additionally we need to capture implicit dependencies due to the first-matching application of rules.

#### 6.3.1 System and Information Flow Models

The input of the information flow analysis is a system model in the form of a directed, symmetric, vertex-typed and -attributed graph. The analysis produces as an output a directed, edge-labeled graph, which we call an information flow graph, as an overlay on the system model graph. Figure 6.1 illustrates our model of a virtualized infrastructure including actors such as administrators. We represent the topology of the virtualized infrastructure with the following graph model.

**Definition 34** (System Model). *Let  $\mathbb{T}$  be a set of vertex types,  $\Sigma$  an alphanumeric alphabet where  $\mathbb{A} \subset \Sigma^+$  is a set of vertex attribute names, and  $\mathbb{D} \subset \Sigma^*$  is a set of attribute values. The system model graph  $G_S = (V_S, E_S, P)$  contains a set of uniquely labeled and typed vertices  $V_S \subset \mathbb{V} := (\Sigma^+ \times \mathbb{T})$ , a set of edges  $E_S \subseteq (V_S \times V_S)$ , and a vertex properties set  $P \subset \mathbb{P} := (\mathbb{A} \times \mathbb{D})$ . A vertex  $v$  is a tuple of vertex label and type  $(l, t) \in V_S$ , and we write  $v.t$  to obtain the type of a vertex. The edges are directed and symmetric, i.e., for each edge  $e = (v_i, v_j)$  there must exist an edge  $e' = (v_j, v_i)$  in  $E_S$ . A partial function  $\text{attr} : (V \times \mathbb{A}) \dashrightarrow \mathbb{D}$  is defined as an attribute function which returns for a given vertex and attribute name the attribute value. We also use the notation  $v.a$  for  $a \in \mathbb{A}$  to obtain the attribute value instead of  $\text{attr}(v, a)$ .*

---

**Definition 35** (Hierarchically-Typed and Relational Vertex Model). *The vertex types  $\mathbb{T}$  form a tree hierarchy that establishes a partial ordering of the types based on transitive parent-child relations, i.e., child  $<$  parent or a directed edge (child, parent), where the root node type is called **Any**. We define a type relation  $T \subset (\mathbb{T} \times \mathbb{T})$ . A given type pair  $(t_i, t_j) \in (\mathbb{T} \times \mathbb{T})$  is considered adjacent if there exists a pair  $(t'_i, t'_j) \in T$  for which  $t_i \leq t'_i$  and  $t_j \leq t'_j$ .  $G_S$  is considered valid if  $\forall (v_i, v_j) \in E_S : \text{adjacent}(v_i.t, v_j.t)$ .*

The data model of the system is a simplified form of the *Enhanced Entity-Relationship Model* that establishes sub-typing and relationship modeling.

**Definition 36** (Information Flow Model). *The information flow model graph  $G_I(G_S) = (V_S, E_I)$  is derived from  $G_S$  and contains the set of typed and attributed vertices  $V_S$  of the system model graph  $G_S$ , as well as a set of directed and labeled edges  $E_I \subseteq (V_S \times V_S)$  with an edge label function  $f : E \rightarrow \{\text{flow}, \text{noflow}\}$ . An edge  $e = (v_i, v_j)$  with label flow means that information from  $v_i$  can flow to  $v_j$ , whereas noflow indicates no flow. We write information flow edges in short form as *iedge*.*

In terms of dynamic behavior of the models, we consider the system model graph to be static. In Section 6.4 we will study a fully dynamic system model graph and its implications on our information flow graph. However the information flow model is dynamic, i.e., during application of rules we are inserting new edges.

---

### 6.3.2 Information Flow Rules

---

The information flow rules encode trust and isolation assumptions of system model elements by the user. They are a mandatory input for constructing the information flow graph from the system model graph. The application of rules in a first-matching semantic and the construction of the information flow model is discussed in Section 6.3.4.

**Definition 37** (Information Flow Rule). *Let  $F$  be a set of flow types  $\{\text{flow}, \text{noflow}\}$ ,  $\mathbb{T}$  a set of system model vertex types. A rule  $r$  is a tuple  $r = (ft, t_i, t_j, p_a, p_c)$ , where  $ft \in F$ ,  $t_i, t_j \in \mathbb{T}$ ,  $p_a$  a predicate on attributes of vertices  $V_S$  and  $p_c$  a predicate on connectivity of vertices in  $G_I$ . The rule describes information flow from a vertex of type  $t_i$  to another vertex of type  $t_j$ .*

*A rule is considered simple if  $(t_i, t_j)$  is adjacent (cf. Definition 34) and  $p_c$  is always true. A rule is considered complex if  $(t_i, t_j)$  is non-adjacent and  $p_c$  may only be using **connected** statements on simple flow edges. A default simple rule only operates on type **Any**, is adjacent, and  $p_a$  and  $p_c$  are always true. An information flow edge  $e$  that is later produced by a simple or complex rule will have the rule type  $r t(e)$  of either simple or complex.*

We use rules with connectivity conditions for expressing tunneled information flow between two system components that are not directly connected in the system model. For example in our case study, we use connectivity conditions to model VLANs and other form of tunnels (GRE, VPN) between the tunnel endpoints.

Depending on the flow type of the default rule, the analysis may either tend to produce false positives in case of a default flow because we are over-approximating the possible information flow. In case of a default noflow, the analysis may produce false negatives.

---

#### 6.3.2.1 Attribute and Connectivity Conditions

---

Our definition of an information flow rules includes two predicates  $p_a$  for attribute conditions and  $p_c$  for connectivity conditions. We treat those predicates separately and do not allow mixing attribute with connectivity conditions. The predicates are expressed in Boolean algebra.

The *attribute* predicate  $p_a$  takes two vertices  $v_i, v_j$  and the property set  $P$  of the system model graph. The predicate can use equality expressions on, and only on, the attributes of  $v_i$  and  $v_j$ . We do not allow nested attribute conditions.

The *connectivity* predicate  $p_c$  takes two vertices  $v_i, v_j$  and the information flow graph  $G_I$ . The connectivity conditions is built upon a **connected** predicate that we define as the following: **connected(a,b)** for  $a, b \in V_S$  returns true if there exists a path from  $a$  to  $b$  in the information flow sub-graph  $G_{I,flow} = (V_S, E_{I,flow})$  where  $E_{I,flow} = \{e \mid e \in E_I \wedge f(e) = \text{flow}\}$ . Only *complex* rules are allowed to have connectivity conditions and only on the information flow sub-graph that was produced by *simple* rules, i.e., on the following edge set:  $E_{I,flow,simple} = \{e \mid e \in E_{I,flow} \wedge rt(e) = \text{simple}\}$ . As the flow edges are directed, **connected** is not necessarily symmetric. The vertex parameters of **connected** can either be  $v_i$  and  $v_j$ , or adjacent vertices of those. We call a condition predicate *closed* if it has been partially applied with the two vertices  $v_i, v_j$ . The resulting closure still takes either the current attribute property set  $P$  for attribute conditions or the current  $G_I$  for connectivity conditions.

---

### 6.3.2.2 Rule Matching and Evaluation

---

Given an information flow rule and a pair of vertices, we define when a rule matches and what the evaluation of that rule returns.

**Definition 38** (Rule Matching and Evaluation). *Given a rule  $r = (ft, t_i, t_j, p_a, p_c)$  and a pair of vertices  $v_i$  and  $v_j$ . The current system state is given as  $G_S = (V_S, E_S, P)$  and  $G_I(G_S)$ . A rule has a full match if the types match:  $(v_i.t \leq t_i) \wedge (v_j.t \leq t_j)$ , and the conjunction of conditions is true:  $p_a(v_i, v_j, P) \wedge p_c(v_i, v_j, G_I)$ . The rule returns an information flow edge  $e = (v_i, v_j)$  with flow label  $f(e) = ft$ . If the types do not match, then the rule evaluates to nil. If the types match, but any of the predicates does not, then we have a partial match, and we return an implicit dependency, which contains the rule, the closed attribute and connectivity condition predicates, as well as the vertex pair.*

Here we only introduced the matching of a single rule and the possible in return values. In Section 6.3.4 we discuss a first-matching algorithm that takes a set of well-ordered rules for evaluation.

---

### 6.3.2.3 Attribute and Connectivity Dependencies

---

If a rule fully matches and returns an information flow edge, this edge depends on the rule's attribute and connectivity condition. To prepare the grounds for the dynamic system model analysis, we associate these dependencies with the edges.

An *attribute* dependency **AttrDep** is a set of tuples  $(v, a)$ , where  $v \in V_S$ ,  $a$  is an attribute of vertex  $v$ , and a predicate  $d_a$ , which is the closed attribute predicate that is true if the attribute dependency is still fulfilled. Each usage of a vertex attribute in the attribute condition will result in a vertex-attribute tuple in the resulting attribute dependency. Similarly, a *connectivity* dependency **ConnDep** is a set of tuples  $(v_i, v_j, p)$  where  $v_i$  and  $v_j$  are the connectivity endpoints and  $p$  an optional connectivity path. Predicate  $d_c$  indicates if the connectivity dependency is still fulfilled, which is again the closed connectivity predicate.

---

### 6.3.3 Ordering of Information Flow Rules

---

The ordering of rules is important since we apply them in a first-matching semantic in our analysis. In this section we discuss how to establish a partial ordering for a given sequence of rules based on the rules' types and conditions. We derive a directed acyclic graph, the *Rule Order Graph*, from the partial ordering, which yields a rule evaluation order for the analysis.

For a sequence of rules  $R$  we define a function  $\text{cmp} : (R \times R) \rightarrow \{\text{EQ}, \text{LT}, \perp\}$  that establishes a partial ordering for any pair of rules with the return values less-than (LT), equal (EQ), and undefined ( $\perp$ ).

**Table 6.1.:** Subset of Information Flow Rules Relevant for PortGroup (PG) VLAN Isolation.

#	Kind	Flow	Directed Node Pair	Condition(s)	Edge Dependency
1	Simple	stop	VSwitch → PortGroup	$PG.vlanId \neq 0$	Attribute VLAN ID
2	Simple	stop	PortGroup → VSwitch	$PG.vlanId \neq 0$	Attribute VLAN ID
3	Simple	follow	Any → Any	—	—
4	Complex	follow	PortGroup → PortGroup	$PG_i.vlanId \neq 0 \wedge PG_i.vlanId = PG_j.vlanId$ $\wedge \text{connected}(vswitch(PG_i), vswitch(PG_j))$	Attribute VLAN ID Connectivity of VSwitches
5	Complex	stop	PortGroup → PortGroup	—	—

We use a running example to illustrate the rule ordering. We defined a subset of rules in Table 6.1, which are derived from our case study description from Section 6.2. We establish the ordering using two implementations of the *cmp* function: one for type-based and another for condition ordering. If type-based ordering returns equality for a given rule pair, we need to further order by conditions.

### 6.3.3.1 Type Ordering

Given our type hierarchy from Definition 35, two rules may operate on different levels of this hierarchy. In general, given two rules that operate on types that are in a transitive parent-child relationship, then the rule with the child types has to be evaluated first. Otherwise, the more general parent-type rule is always applied. We define the *cmp* function for type-based ordering as the following:

$$\text{cmp}_{\text{type}}(r_1, r_2) = \begin{cases} \text{EQ} & \text{if } r_1.t_i = r_2.t_i \wedge r_1.t_j = r_2.t_j \\ \text{LT} & \text{if } (r_1.t_i < r_2.t_i \wedge r_1.t_j \leq r_2.t_j) \vee (r_1.t_i \leq r_2.t_i \wedge r_1.t_j < r_2.t_j) \\ \text{ERR} & \text{if } (r_1.t_i < r_2.t_i \wedge r_1.t_j > r_2.t_j) \vee (r_1.t_i > r_2.t_i \wedge r_1.t_j < r_2.t_j) \\ \perp & \text{otherwise} \end{cases}$$

The types of the type tree form a partial order and here we establish a product order for tuples of two types. LT if one type is strictly less than the other in the tuple. We have an error case (ERR) if we have conflicting relations, where one type in the tuple is strictly less but the other one is strictly greater than the corresponding type of the other tuple. In any other case the ordering is undefined.

In case of EQ, we require further condition-based ordering. In case of ERR we need to abort the information flow analysis as the rules ordering is inconsistent. If any of the pair types are not in a (transitive) parent-child relation, then the ordering is undefined, i.e., we obtain a partial order of the rules based on their types. If for all rules the node type pairs are distinct/non-relational, then the rules are confluent, i.e., the order of which they are evaluated does not matter.

### 6.3.3.2 Condition Ordering

For the rules that are defined for equally typed nodes we require ordering based on their condition predicates. Basically when given two equally typed rules, the rule with the most specific condition has to be first, and the one with the most general condition last. The two rules must not be equal, but can be either in a LT or  $\perp$  relation.

Given two rules  $r_1$  and  $r_2$  with their condition predicates  $p_1 = r_1.p_a \wedge r_1.p_c$  and  $p_2 = r_2.p_a \wedge r_2.p_c$ . The predicate atoms are `connected` statements and attribute equalities on typed constants and variables. We need to define a partial order (EQ, LT,  $\perp$ ) returned by a function  $\text{cmp}_{\text{cond}}$  analog to the type-based ordering. We employ an approach based on the interpretation of predicates and truth assignments. We



leverage existing work in this areas, e.g., for the ordering of methods with conditions [EKC98] or predicate interpretation with parametrized truth assignments and value domains [AU95].

A function `truths` is defined that returns for two given predicates a list of variable assignments from the value domains as well as connected statement truth assignments. Further, we define a `eval` function that takes a truth assignment (variable value assignments and connected truth assignments) and a predicate, and returns either true or false for the predicate evaluation under the given variable assignments and connected statement assignments. The function substitutes variables with assigned values and connected statements with truth assignments. The predicate with substitutions is then evaluated.

### Connectivity Truth Assignments

We have a set  $C$  of `connected(a,b)` statements each for a unique pair of vertices  $(a,b)$ . Each unique statement is considered true or false leading to  $2^{|C|}$  possible truth assignment combinations. Note that the statement is not symmetric, i.e., when `connected(a,b)` is assumed true it does not mean that `connected(b,a)` must be true too.

### Attribute Equality Truth Assignments

We write attribute equalities as a predicate `AttrEq(a1,a2)` for  $a1 = a2$ . We have a set of `AttrEq(a1, a2)` where the parameters can either be constants or variables of types integer, string, or Boolean. The *domain* of integers  $D_i$  includes the integer constants and for each variable a unique random integer value. Analog are the domains  $D_s$  for strings and  $D_b$  for Boolean values. We consider the set of attribute variables typed integer  $A_i$ , string  $A_s$ , or Boolean  $A_b$ . The possible combinations of value assignments for the attribute variables are  $|D_i|^{|A_i|} \cdot |D_s|^{|A_s|} \cdot |D_b|^{|A_b|}$ .

### Determining a Partial Ordering of Predicates

The combinations of truth assignments for `connected` statements and value assignments for attribute variables leads to the overall number of combinations:  $2^{|C|} \cdot |D_i|^{|A_i|} \cdot |D_s|^{|A_s|} \cdot |D_b|^{|A_b|}$ . It is exponential to the number of connected statements as well as attribute equality parameters.

Given two rules  $r_1$  and  $r_2$  and their predicates  $p_1 = r_1.p_a \wedge r_1.p_c$  and  $p_2 = r_2.p_a \wedge r_2.p_c$ . We compute the truth and variable assignments with `truths` on those predicates and then iterate over the assignments. For each assignment `eval` evaluates both predicates. We obtain equality if for all assignments the interpretations of the predicates are simultaneously true. For an LT order we require that for all assignments the first predicate implies the second. Otherwise, the order of the predicates is undefined.

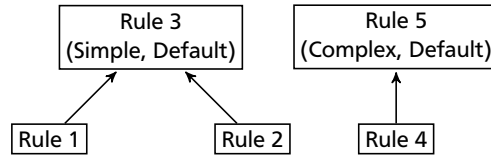
We define the condition-based compare function as the following:

$$\text{cmp}_{\text{cond}}(r_1, r_2) = \text{cmp}'_{\text{cond}}(r_1.p_a \wedge r_1.p_c, r_2.p_a \wedge r_2.p_c)$$

where `cmp'`<sub>cond</sub> is a helper function that operates on the rules' predicates.

$$\text{cmp}'_{\text{cond}}(p_1, p_2) = \begin{cases} \text{EQ} & \text{if } \forall t \in \text{truths}(p_1, p_2) : \text{eval}(t, p_1) \wedge \text{eval}(t, p_2) \\ \text{LT} & \text{if } \forall t \in \text{truths}(p_1, p_2) : \text{eval}(t, p_1) \rightarrow \text{eval}(t, p_2) \\ \perp & \text{otherwise} \end{cases}$$

The two predicates are equal if they are simultaneously true for all truth assignments. In negated form, they are “mutually exclusive exactly if one implies the negation of the other” [EKC98] or in other words a NAND operation. They are LT if the specific predicate  $p_1$  always implies the more generic predicate  $p_2$ , i.e., when  $p_1$  is true then  $p_2$  must be true, which is a logical implication. Ernst [EKC98] uses the term “overrides” to describe that one predicate is a specialization of another, i.e., the specific predicate overrides the general predicate. He defines it as “Method  $m_1$  overrides method  $m_2$  iff  $m_1$ 's predicate implies that of  $m_2$ , that is, if (not  $m_1$ ) or  $m_2$  is true.” [EKC98].



**Figure 6.2.:** Order Graph of Rules of Table 6.1 based on Type and Condition Ordering.

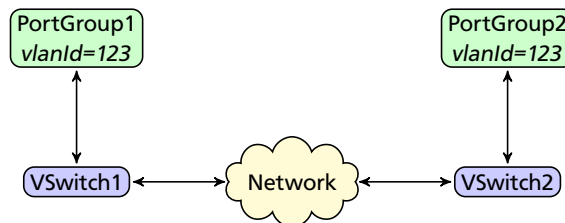
### 6.3.3.3 Establish a Rule Order Graph

We establish a *Rule Order Graph*  $G_R$  as a directed graph with rule identifiers as vertex labels. A directed edge  $(r_1, r_2)$  with edge head  $r_1$  and edge tail  $r_2$  represents that  $r_1 < r_2$ . It means that  $r_1$  must be evaluated before  $r_2$ . The relation between partial orders and DAG (also later topological sorting) is well known and we use it here.

Our example rule set produces the following rule order graph illustrated in Figure 6.2. The simple rules Rule 1 and Rule 2 are dependent on the default simple rule Rule 3. Rule 1 and Rule 2 are however unrelated in terms of the type pair they operate on, because  $VS \not\leq PG$  and vice versa. Rule 4 is a complex and non-adjacent rule, therefore independent of all the other simple rules, but dependent on the default complex rule Rule 5.

### 6.3.4 Application of Rules and Construction of Information Flow Model

The construction of the information flow model is based on the system model and uses the matching of individual rules (cf. Section 6.3.2.2) from a well-ordered set of rules (cf. Section 6.3.3). Hence, the application of rules requires as inputs the system model graph as well as the rule order graph. The output is an information flow model with the edge dependencies derived from the rule application. For our case-study we use both the system model sub-graph as illustrated in Figure 6.3 and the subset of rules shown in Table 6.1. The final output is shown as an overlay graph in Figure 6.4.



**Figure 6.3.:** Input Model: Subset of system model graph

The algorithm uses a first-matching semantics of the rules. A topological sort of the rule order graph provides a valid evaluation order that adheres to the ordering of the rules. If the topological sort cannot produce a sorting we will report an error. The topological sort may produce many valid evaluation orders, because two rules with undefined ordering are confluent and can be evaluated in any order. The application of the rules is defined in Algorithm 1. The key points of the algorithms are the following:

- *TopoSort* performs a topological sorting and produces a linear ordering  $R$  of the rules in the rule order graph  $G_R$ . We split the rule sequence  $R$  into simple (adjacent) rule set  $R_{\text{simple}}$  and a complex (non-adjacent) rule set  $R_{\text{complex}}$ . For the simple rules we iterate over the edge set of the system model graph. For the complex rules we obtain the nodes for the matching (sub-)types using a function *TypedNodes* and evaluate the node pairs.
- We have a function *EvalRule* that tries to match a rule (cf. Section 6.3.2.2) and returns either an information flow edge (*iedge*), implicit dependency, or *nil* for a given rule, node pair, as well as

---

**Algorithm 1:** Application of Information Flow Rules.

---

```
Data: Rule Order Graph  $G_R$ , System Model Graph  $G_S = (V_S, E_S, P)$   
Result: Information Flow Graph  $G_I = (V_I, E_I)$  and Components Graph  $G_C$   
// Initializing the information flow graph and compute rule order  
 $G_I \leftarrow (V_S, \emptyset)$   
 $R \leftarrow \text{TopoSort}(G_R)$   
 $D_S \leftarrow \emptyset$  // Initializing implicit dependencies of simple rules  
// Processing simple rules of  $R$   
foreach  $(u, v) \in E_S$  do  
  | foreach  $r \in R_{\text{simple}}$  do  
  |   |  $\text{res} \leftarrow \text{EvalRule}(r, u, v, G_S, G_I)$   
  |   | if  $\text{res}$  is iedge then  
  |   |   |  $(D_S, d) \leftarrow \text{ImplicitDeps}(\text{res}, r, D_S)$   
  |   |   |  $\text{iedge.implicit} \leftarrow d$   
  |   |   |  $E_I \leftarrow E_I \cup \{\text{res}\}$   
  |   |   | break  
  |   | else if  $\text{res}$  is implicit dependency then  
  |   |   |  $D_S \leftarrow D_S \cup (r, \text{res})$   
  | if  $D_S \neq \emptyset$  then  
  |   | ERROR // Unclaimed implicit dependencies  
// Updating the information flow graph and compute components graph  
 $G_I \leftarrow (V_S, E_I)$   
 $G_C \leftarrow \text{ComputeSCC}(G_I)$   
 $D_C \leftarrow \emptyset$  // Initializing implicit dependencies of complex rules  
// Processing complex rules of  $R$ .  
foreach  $r \in R_{\text{complex}}$  do  
  |  $V_{r,i} \leftarrow \text{TypedNodes}(r.t_i, V_S)$   
  |  $V_{r,j} \leftarrow \text{TypedNodes}(r.t_j, V_S)$   
  | foreach  $(u, v) \in V_{r,i} \times V_{r,j}$  do  
  |   |  $\text{res} \leftarrow \text{EvalRule}(r, u, v, G_S, G_I, G_C)$   
  |   | if  $\text{res}$  is iedge then  
  |   |   |  $(D, d) \leftarrow \text{ImplicitDeps}(\text{res}, r, D_C[(u, v)])$   
  |   |   |  $D_C[(u, v)] \leftarrow D$   
  |   |   |  $\text{iedge.implicit} \leftarrow d$   
  |   |   |  $E_I \leftarrow E_I \cup \{\text{res}\}$   
  |   |   | break  
  |   | else if  $\text{res}$  is implicit dependency then  
  |   |   |  $D_C[(u, v)] \leftarrow D_C[(u, v)] \cup (r, \text{res})$   
  | if  $D_C \neq \emptyset$  then  
  |   | ERROR // Unclaimed implicit dependencies  
// Final information flow graph and compute components graph  
 $G_I \leftarrow (V_S, E_I)$   
 $G_C \leftarrow \text{ComputeSCC}(G_I)$   
return  $(G_I, G_C)$ 
```

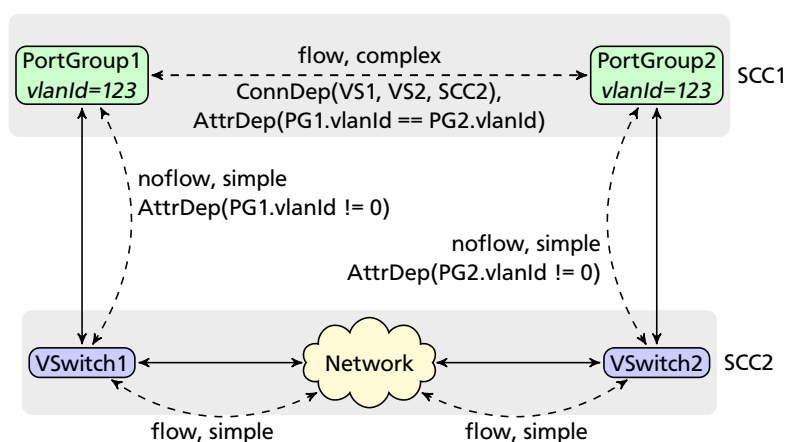
---

system and information flow models and optional component graph. A pre-condition of the rule evaluation is that  $(u, v) \notin E_I$ .

- For a returned information flow edge, we find, obtain, and remove the implicit dependencies of previous rules using the *ImplicitDeps* function (cf. Section 6.3.4.1). We associate the implicit dependencies with the *iedge*, i.e., setting *iedge.implicit*s, and insert the edge into the information flow model graph.
- If an implicit dependency is returned, we store the dependency together with the rule and for complex rules also together with the evaluated node pair. In the algorithm  $D_S$  holds the implicit dependencies of simple rules, and  $D_C$  holds the ones of complex rules indexed by the evaluated node pairs.
- In case of *nil* we simply evaluate the next rule. We report an error if we have unclaimed implicit dependencies, which have not been taken by another rule. For example no default rule for adjacent rules, or no catching rule for the non-adjacent rules.

### Strongly Connected Components

As an optimization to determine reachability, we use strongly connected components (SCC) and a component graph (or also called reachability tree) [CSRL01, Section 22.5]. They allow us to efficiently evaluate connected statements in a rule's connectivity condition. A component graph is a DAG that contains the SCCs as vertices and there exists a directed edge between two SCCs if there exists a directed edge between any two elements that are contained in the respective SCCs. Elements within one SCC are mutually reachable, i.e., reachability can be checked by set membership. To determine reachability between two elements that are not part of the same SCC, we try to find a path in the component graph between their respective SCCs. The reachability is by definition only unidirectional. With the function *ComputeSCC* we compute the SCCs and the component graph on a sub-graph of  $G_I$  that only contains flow-labeled edges and is further parametrized by the rule type. During rule application we compute the SCCs [Tar72] after all simple rules have been evaluated, because the subsequent complex rules may have connectivity conditions based on the result of the simple rules. Finally we also compute SCCs after the complex rules have been evaluated for the entire information flow graph, in order for policy checks to verify connectivity.



**Figure 6.4.:** Output Model: Graph model annotated with dashed information flow edges of different kinds (simple, complex, attribute- and connectivity-dependent).

---

### 6.3.4.1 Implicit Dependencies

---

To further prepare the ground for the dynamic information flow analysis, we need to record when a rule matched because a child rule (in the rule order graph) did not match due to mismatching attribute or connectivity conditions. A resulting information flow edge from a rule evaluation obtains the negative conditions from the rule's child rules. It means the result only exists because one of the child rules did not match. Once the system or information flow model are changing a previous rule may match and the current result needs to be invalidated.

The function *ImplicitDeps* takes an *iedge*, a rule, and a list of implicit dependencies. It returns a new dependency list with the matching ones removed and a disjunction of the matching dependencies' predicates. The matching dependencies for a rule  $r$  are:  $D' = \{(r', d_a, d_c) \in D \mid r' < r\}$  with  $r' < r$  for a transitive order in the rule order graph (cf. Section 6.3.3.3). The remaining implicit dependencies are simply  $D \setminus D'$ . In the case of complex rules, we need to further index the implicit dependencies by the vertex tuple. The implicit dependencies  $D'$  are then associated with the *iedge*. An *iedge* is valid if it's own dependencies are true, but not any implicits. Given  $D' = (i_1, \dots, i_k)$ , then  $iedge.implicit(P, G_I) = \bigvee (i_j.d_a(P) \wedge i_j.d_c(G_I))$ . Both the closed attribute and connectivity predicates have to be true for a child rule to match. If any of the child rules match then the parent rule must be invalidated, therefore the disjunction of implicit dependency predicates. The validity of an information flow edge under the current property set and information flow graph is given as a predicate  $valid(iedge, P, G_I) = iedge.d_a(P) \wedge iedge.d_c(G_I) \wedge \neg(iedge.implicit(P, G_I))$  where  $d_a$  and  $d_c$  are again the closed condition predicates. We derive the attribute and connectivity dependencies not only for the rule's own conditions, but also for its implicits.

---

### 6.3.5 Algorithm Analysis

---

We analyze our approach with regard to the termination and complexity analysis of the algorithm. We are adapting and extending a firewall fault model and analyze how our analysis prevents such faults. Finally we discuss the correctness of the ordering and application of rules.

---

#### 6.3.5.1 Algorithm Termination and Complexity

---

The termination of algorithm 1 is given by the following properties:

- *Finite Sets*: We apply a finite set of flow rules. We evaluate the simple rules by iterating over the finite edge set  $E_S$  and similarly for the complex rules on subsets of vertex products  $V_S \times V_S$ . In each iteration of algorithm 1 the set of *non-evaluated* edges or vertex pairs shrinks, and we do not modify these sets during the execution of the algorithm.
- *Limited Inter-Rule Dependencies*: We do not have any circular or self dependencies among rules, which could otherwise result in a rule application without termination. Rules depend on node types and their attributes, which are not influenced by any other rules. Complex rules may depend on connectivity, which is influenced by other rules. However, we limit connectivity conditions to only *iedges* produced by simple rules (cf. Definition 37). This provides a one-way dependency from complex to simple rules, but no circular or self dependencies exists among the complex rules.
- *Termination of Helper Functions*: The helper function *EvalRule* performs a rule matching and only uses a terminating BFS on the finite component graph. *ImplicitDeps* is iterating over the finite set of implicit dependencies. *TypedNodes* returns a subset of nodes from the finite nodes set  $V_S$ . *TopoSort* and *ComputeSCC* are existing algorithms and are variants of DFS, which is terminating for finite graphs with node coloring.

We analyze the run-time complexity of our analysis by breaking it up into the following steps:

- *Rules Ordering and Topological Sort:* Type-based rule ordering requires  $|R|^2$  comparisons, each comparison's complexity linear to the depth of the type hierarchy. Condition-based ordering requires  $|R|^2 \cdot 2^{|C|} \cdot |D_i|^{|A_i|} \cdot |D_s|^{|A_s|} \cdot |D_b|^{|A_b|}$  comparisons, which is exponential to the number of connected statements and attribute equalities. Topological sort is  $O(|V_R| + |E_R|)$ , since it is based on DFS. The ordering and topological sort is done once for a given rule set.
- *Rule Evaluation:* *EvalRule* is constant with regard to  $G_S$  and  $R$ , but for a given rule  $r$  it depends on the number of condition statements in  $p_a$  (the sets of integer/string/boolean attribute atoms:  $|A_i| + |A_s| + |A_b|$ ) and  $p_c$  (the set of connected statements  $|C|$ ). Connected statements can be evaluated in constant time using set membership checks when the nodes in the same SCC, or linear to the size of the component graph by using path finding (e.g., BFS). Attribute condition atoms are evaluated in constant time.
- *Implicit Dependency:* For simple rules the implicit dependencies are derived linearly to the size of the rules. For complex rules, we first perform a constant lookup with the node pair  $(u, v)$  followed by finding the implicit dependencies in linear time in the rule set.
- *Simple Rules Application:* The evaluation of the simple rules requires  $|E_S| \cdot |R_{\text{simple}}|$  applications of *EvalRule* and *ImplicitDeps*.
- *Complex Rules Application:* The evaluation of the complex rules requires  $|R_{\text{complex}}| \cdot |V'_S|^2$  applications of *EvalRule* and *ImplicitDeps*. We evaluate the complex rules for the matching pairs of typed nodes, where  $V'_S$  is a subset of  $V_S$ , which practically makes a difference but not asymptotically.
- *Computation of Strongly Connected Components:* We compute the strongly connected components (SCCs) twice: first after the application of simple rules and a second time after the complex rules application. Tarjan's algorithm [Tar72] to compute SCCs has a complexity of  $O(|V_S| + |E_T|)$  as well as the component graph creation is linear [CSRL01, Section 22.5].

In summary, we can differentiate between load-time and run-time complexity. Load-time is concerned about rule ordering and run-time about the evaluation of rules. The dominating parts for load-time are the quadratic rule complexity for rule ordering in general and exponential condition ordering in particular. In practice this is not a problem, because we only do it once for a given rule set, the condition ordering only happens for equally node-typed rules, and the number of connectivity and equality statements in the predicates is small. During run-time the dominating factor is the evaluation of complex rules with quadratic vertex set size.

In terms of space complexity, the upper bound is given by a full mesh information flow graph given by  $|V_S|^2$  iedges. Each iedge may obtain and store implicit dependencies from  $|R| - 1$  rules. The component graph based on the SCCs would as the upper bound contain  $|V_S|$  SCCs, where each system model vertex is its own SCC. As a future optimizations, we can introduce information flow *vertices* in addition to the iedges. Such a vertex would allow to move from a full mesh information flow graph to a star topology. For instance in the case of Rule 5 of Table 6.1, which currently would create a full mesh between the non-matching portgroups, we can create one information flow vertex for the *noflow* portgroups.

### Complexity Comparison with Traversal Analysis

We now compare the complexity of the information flow graph based analysis, denoted as `FLOWGRAPH` approach, to the graph traversing analysis of Chapter 3, which we denote as the `TRAV` approach. The `TRAV` information flow analysis performs a graph traversal based on a set of traversal rules, which are similar to our flow rules, starting from a set of information source vertices. Each source obtains a unique *color* that propagates through the graph based on the rules' decisions. In the worst case, we have  $|V_S|$  information sources and perform a DFS-like traversal with  $O(|V_S| + |E_S|)$  from each source, leading to a dominating

---

run-time complexity of  $|V_S|^2$ . In terms of space complexity, each vertex has to store at least  $|V_S|$  unique colors, although rules may create an arbitrary number of “sub-colors”, also called tags. We compare the two approaches for the following steps:

- *Rules Ordering*: This step is required by both approaches in similar complexity, although in TRAV a good ordering is assumed and no automated ordering performed. The rules conditions of TRAV do not contain connected statements, which simplifies the condition ordering to just attribute and tags-based conditions.
- *Rule Evaluation*: Considering a single rule evaluation, both approaches operate on the vertices attributes. FLOWGRAPH also allows connectivity conditions that is evaluated linear to the size of the component graph in the case of a complex rule. The TRAV approach operates on the current color and sub-colors, which is independent of the graph size.
- *Rule-Dependent Metadata*: Rules can lead to metadata that needs to be stored in addition to the information flow state. In FLOWGRAPH we store explicit and implicit dependencies. Similarly, the TRAV rules can create sub-colors or tags. In both cases the space complexity of the metadata is highly dependent on the specific rules set.
- *Rules Application*: In FLOWGRAPH we differentiate between simple and complex rules. We do not have this differentiation in the TRAV approach, but only have simple rules equivalents. For complex rules, the dominating run-time complexity is given by  $|V_S|^2$ , although in practice we evaluate a vertex subset, e.g., only portgroups. In the TRAV approach we start for instance from VMs as information sources, and typically  $|PG| \ll |VM|$ .
- *Connectivity Evaluation*: In FLOWGRAPH we use SCCs for efficient connectivity evaluation, which takes in the worst case linear to the component graph size. In TRAV we evaluate connectivity based on colors, which is constant for a color set membership. However the connectivity depends also on the information sources, i.e., we can evaluate connectivity only from specific information source vertices.

In summary, the dominating run-time complexity of  $|V_S|^2$  is given in this approach by the complex rules evaluation, although in practice we typically evaluate a subset of  $V_S$ . In the TRAV approach the same complexity is dominating when starting a graph traversal from each vertex as an information source. The evaluation and application of rules differs slightly between the two approaches, where this one depends on connectivity conditions and the other one on color tags. In terms of space complexity, both approaches have to store in the worst case  $|V_S|^2$  information flow states, either in the form of a full mesh information flow graph or as  $|V_S|$  colors for each vertex. We outline as part of future optimizations how this space complexity can be reduced by moving from a full mesh to a star topology with *information flow vertices*. Although for the full system mode analysis the both approaches are very similar in terms of run-time and space complexity, in Section 6.4 we compare the two approaches with regard to analyzing a dynamic system model.

---

### 6.3.5.2 Fault Model for First-Matching Rules Application

---

The goal of the information flow analysis is to extrapolate the isolation decisions between system model components by a user to the entire system. Instead of deciding for each individual edge and node pair in the system model graph if there is an information flow or not, the user captures generalized decisions using the flow rules.

The extrapolation is based on the specific rule set and the application of those rules. The first depends on the decisions by the user and there is no clear right or wrong. We can reduce the correctness of the extrapolation to the correctness of the rules. The second depends on the dependencies between simple

---

and complex rules and the ordering of the rules due to our first-matching semantic. The rule application first evaluates the simple rules and only then the complex rules to satisfy their one-way dependency. To analyze the rule ordering, we adapt and extend the firewall fault model [CLHX10, CLHX12], because in firewall rules we also deal with first-matching semantics and similar faults.

- **Wrong Order:** The ordering of rules is critical in a first-matching application and our approach has to ensure a well-ordered set of rules. We have to consider the following cases how rules can be in a wrong order:
  - **Wrong Type Ordering:** A rule with more generic types must appear after a rule with more specific types, i.e., sub-typed rules before super-typed rules. We rely on the facts that the type tree establishes a partial order of the types and product order to establish a partial order for the tuple of types.
  - **Wrong Conditions Ordering:** For two equally typed rules, the rule with the more generic condition must appear after the rule with the more specific condition. We obtain a partial predicate order using truth assignments and interpretation with variable assignments.
  - **Inter-Rule Dependencies:** Rules may only depend on each other due to connectivity conditions. However this could lead to circular dependencies. We prevent cycles by only allowing one-way dependency from complex to simple rules. We always evaluate simple rules first so that the dependency of the complex rules is satisfied.
  - **Conflicting Rules:** When rules operate on the same types and conditions but producing different results. We prevent this by requiring a non-equal ordering after type and condition ordering for two given rules.
- **Missing Rule:** Depending on the default rule, a missing rule may lead to false positives (default is flow) or false negatives (noflow). We can reduce this fault to the correctness of the rules and their coverage (number of explicit vs. default rule, cf. Chapter 3).
- **Wrong Predicates:** Wrong conditions may result in false positives or negatives when a rule triggers under the wrong circumstances or is not triggered at all. This fault is again reduced to the correctness of the rules.
- **Wrong Decision:** A rule may return a wrong flow decision (Flow/NoFlow). This can be a crucial mistake that can also lead to false positives and negatives. In general we advice to perform NoFlow decisions in the simple rules with a default flow decision, in order to mitigate false negatives. In such a case a wrong decision can be spotted more easily.
- **Wrong Extra Rule:** Old rules may remain in the rule set. They could result in ordering problems, which we would detect. However they could also result in false positives or negatives. We do not see this as a major problem as we are dealing with more static and smaller rule sets compared to network firewall configurations.

In summary, many faults can be reduced to the correctness of the rules themselves. In practice, for the different application domains we envision a rule set that is based on best practices. In addition, the ordering of rules is crucial and our approach ensures a well-ordered rules set and a rule application that satisfies inter-rule dependencies.

---

### 6.3.5.3 Correctness of Rule Ordering and Application

---

We have to show the correctness of two parts of the analysis for the static system model case: First, the correct ordering of rules; Second, the correct application of the ordered rules. The ordering of rules relies on the following parts of the analysis, which are build upon existing work:



- *Type Ordering*: The vertex types form a type hierarchy in form of an in-tree, i.e., a rooted tree where all vertices have a unique path to the root. We can derive a partial ordering based on the child-parent edges. A product order establishes a partial ordering for a tuple of partially ordered elements.
- *Condition Ordering*: Using truth assignments and variable assignments from a value domain, we establish a partial ordering of the rule's predicate. In particular two predicates are equal when they are simultaneously true for all assignments, and one predicate is less than another if the first implies the second. Predicate ordering has been used also in other domains [EKC98].
- *Rule Order Graph*: We construct a DAG based on the partial ordering between rules using the type and condition partial ordering. A DAG can represent a partial order, where a directed edge  $(u, v)$  represents  $u \leq v$ . In particular the following properties are fulfilled: *reflexive* since each vertex can reach itself ; *transitive* since we can construct a transitive closure; *asymmetric* because with  $u \leq v$  and  $v \leq u$  if  $u \neq v$  then we would have a cycle, so not a DAG anymore.
- *Topological Sorting*: Given a DAG, the topological sorting produces a linear ordering (out of potentially many valid ones) of the vertices based on their directed edges. This is a well-known algorithm [CSRL01, Section 22.4] used in areas such as task scheduling. In our application, we apply topological sorting on our Rule Order Graph, which is a DAG, to obtain a rule evaluation order.

For the second part we show that an error in the reachability of any two vertices in the information flow model can only be caused by an error in the individual information flow rules, but not by an error in the application of the rules by our algorithm.

- *Completeness of rules application*: For a given rule set, the application of these rules is complete. We evaluate all edges with the simple rules in first-matching semantics. We evaluate all vertex pairs that match or are sub-types of a complex rules.
- *Reduction to the correctness of individual rules*: Given any pair of vertices  $(a, b)$  where  $connected(a, b)$  is true. There must exist a path  $p = [e_1, \dots, e_k]$  of ordered  $k$  iedges with flow type flow, by definition of the connected predicate. An edge  $e_i = (u, v)$  in the path has either been created by a simple rule if there exists an edge  $(u, v) \in E_S$  or by a complex rule in the non-adjacent case. Either rule has made a flow decision. If the expected outcome was that  $connected(a, b)$  is false, then a simple or complex rule returned the wrong flow decision: instead of flow it should have return noflow.

Similarly, if  $connected(a, b)$  is false, given all possible paths  $P$  between  $a$  and  $b$ , then all paths must contain a noflow iedge:  $\forall(p \in P)\exists(e \in p) : f(e) = \text{noflow}$ . If the expectation was that  $connected(a, b)$  is true, then there must exist one path for which all iedges are flow. At least one rule made a wrong flow decision by returning noflow.

---

### 6.3.6 Summary

---

We lay the foundation for the dynamic information flow analysis by introducing a rule-based construction of an information flow graph for a static system model. We defined both the system and information flow models as graphs, introduced the information flow rules and their ordering, and shown an algorithm for the application of such rules. Overall the key concepts of our approach are the following:

- *System and Information Flow Models as Graphs*: The system is modeled as an directed, symmetric, vertex typed and attributed graph. The vertex types form a tree-like type hierarchy and relationships between vertices are modeled. The information flow model is an overlay on the system model (i.e., using the same vertex set) but edge-labeled with flow and noflow as well as directed.

- *Flow Rules and Matching:* The rules capture isolation and trust assumptions of the user into system model components. Based on a vertex pair types as well as attribute and connectivity conditions, the rule returns either a flow or noflow decision. A rule matches a given pair of vertices of the system model when the types are equal or subtypes of the rule, and when the conditions are true. Important for the dynamic analysis is that we record for each edge the attribute and connectivity dependencies, as well as implicit dependencies due to the first-matching rule application.
- *Rule Ordering and Application:* The rules are applied in a first-matching way. Therefore the rules ordering is crucial. We establish a partial ordering based on rules' types and – if equally typed– also on conditions. On the resulting *Rule Order Graph* we perform a topological sort which yields an evaluation order. We always evaluate first the simple rules then the complex ones due to possible connectivity dependency.

---

## 6.4 Fully Dynamic Information Flow Analysis

---

We lay the foundation for the dynamic information flow analysis in the previous section, in particular by recording for the created information flow edges the condition dependencies as well as implicit dependencies from preceding rules that did not match. If connectivity or attributes change, the affected information flow edges with their dependencies have to be re-validated and if necessary partially re-computed.

In this section we discuss the handling of a fully dynamic system model and the implications on the information flow model. Instead of performing an information flow analysis always from scratch when the system model graph changes, we perform an analysis that updates the information flow graph. First we define a change to the system model as a graph delta.

**Definition 39** (System Model Change). *Given a system model graph  $G'_S = (V'_S, E'_S)$ . We define a system model change as a graph delta  $\Delta = (V^+, V^-, E^+, E^-, M)$ , where  $V^+ \subset \mathbb{V}$ ,  $V^- \subseteq V'_S$ ,  $E^+ \subset \mathbb{E}$ ,  $E^- \subseteq E'_S$ ,  $M \subset (V'_S \times \mathbb{A} \times \mathbb{D})$ . The delta contains creator as well as eraser nodes and edges, and a set of node attribute modifiers (vertex, attribute, value).*

We rely on an existing system (cf. Chapter 7) that provides us such system model changes for our case study.

Given a delta  $\Delta$  and two versions of the system model graph: before the change  $G'_S = (V'_S, E'_S)$  and after the change  $G_S = (V_S, E_S)$ .  $G_S$  is constructed from the given  $\Delta$  and  $G'_S$  in the following way:  $V_S = (V'_S \setminus V^-) \cup V^+$ ,  $E_S = (E'_S \setminus E^-) \cup E^+$ , and applying the node modifiers on  $V'_S$ . A *differential* information flow analysis computes an information flow graph  $G_I$  based on an information flow graph  $G'_I$  of the previous version of the system model graph  $G'_S$  and  $\Delta$ . Thereby it operates on the *difference* between the system models given as  $\Delta$  to compute the updated information flow model graph  $G_I$ .

The challenge we solve is to maintain an information flow graph, which is build from simple as well as attribute and/or connectivity-dependent information flow edges, even when connectivity or attributes are changed. Our differential analysis works in two phases: First, given a graph delta, we process the changes to the system model graph and the impact on the information flow graph. In particular applying flow rules for new vertices and edges, removing affected edges while removing vertices and edges, and determining attribute dependency violations due to attribute changes. Second, based on the changes to information flow model in the first phase, we compute and process connectivity changes, in particular determining connectivity dependency violations.

---

### 6.4.1 Translating System Model Changes to Information Flow Changes

---

In the first phase we process the system model graph delta, and for each element of the graph delta  $\Delta = (V^+, V^-, E^+, E^-, M)$  compute information flow graph changes.

- **Node Attribute Changes:** Find all affected attribute-dependent edges and remove them if they are invalid, i.e., their attribute condition does not hold anymore or one of the implicit dependencies is true. Re-evaluate the vertex pairs of the removed iedges and insert potentially new iedges due to re-evaluation.
- **Eraser Edges:** For each system model edge we remove the corresponding information flow edge. Further, if one of the edge's vertices is part of a connectivity-dependent iedge with another vertex as a connectivity endpoint, then the iedge is invalid if the removed edge provides the relation between the vertex and endpoint.
- **Eraser Nodes:** Remove the nodes as well as all their incoming and outgoing edges from the information flow graph. Find all connectivity-dependent edges that require the erased nodes as connectivity endpoints, and remove those edges too.
- **Creator Nodes:** For each node evaluate the complex rules (given the new node, and all the existing matching typed nodes, as well as vice versa), which may create new information flow edges, and insert the created edges into the information flow graph.
- **Creator Edges:** Evaluate simple rules for each new edge and insert the resulting information flow edges in the graph. Analog to edge removal, we need to find the (negative) connectivity-dependent iedges, where the new edge establishes the relation between the vertex and its connectivity endpoint.

Regarding the ordering of the graph delta processing, deletion and modification of vertices can only be performed on the existing vertices of the system model graph as defined in Definition 39. We first perform the node attribute changes, then the deletion of edges and vertices. The final step is the creation of nodes and edges.

Examples of system model changes for our case study are the following and illustrated in Figure 6.5. In case of an attribute change where `PortGroup1`'s VLAN ID changes to zero (denoted as  $\Delta_{VLAN}$ ), the edges between the vswitch as well as the other port group are removed, but a new flow edge is introduced between the vswitch. If we have a node removal, i.e., `VSwitch` is removed from Figure 6.4 (denoted as  $\Delta_{VSwitch}$ ), the edges to `PortGroup1` and `Network` are removed. Additionally, the edge between the port groups is removed, because it is dependent on the connectivity of the vswitches.

In summary, removal of graph elements directly impacts the associated iedges, but also the connectivity-dependent iedges that rely on a removed element as part of their connectivity endpoints. Attribute changes affect the attribute-dependent iedges and require a rule re-evaluation when an iedge's attribute dependencies are violated. For new graph elements we simply evaluate them with our rule set.

---

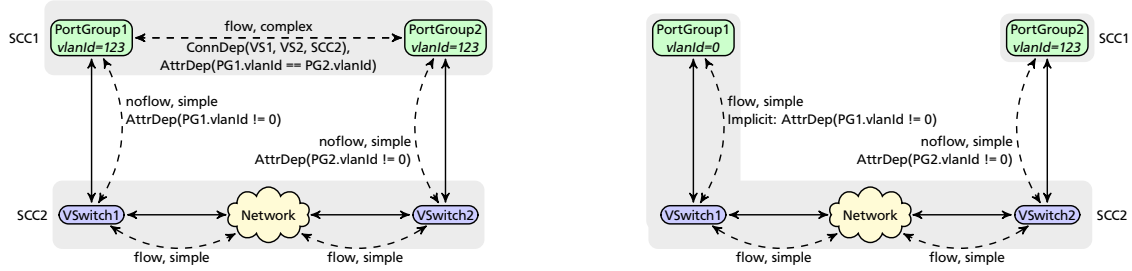
## 6.4.2 Processing Connectivity Changes

---

In the previous phase of our dynamic information flow analysis we processed the system model changes and modified the information flow graph accordingly. The addition and removal of information flow edges may cause changes in the overall connectivity which we have to process and handle as well. In particular we need to handle connectivity-dependent iedges. Only the iedges that have been created by complex rules can be affected, as only they have connectivity dependencies.

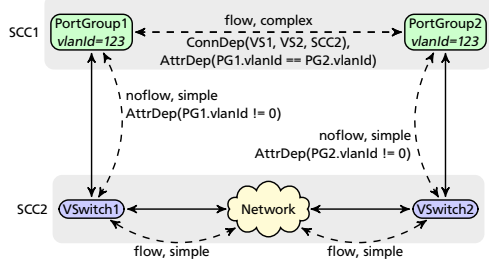
We need an interface that notifies us about connectivity changes, in particular if there exists increased or reduced connectivity, and if any existing connectivity paths in dependencies are affected. Since we are using strongly connected components (SCCs) for efficient connectivity checks, we also use SCC re-computations to tell us after inserting a set of edges, which SCCs have been added/removed, and which edges have been added/removed in the component graph.

With Tarjan's algorithm [Tar72], we can do set operations between an old and a new component graph's vertex and edge sets. Ideally, we would use a dynamic reachability approach [RZ04] that can also provides

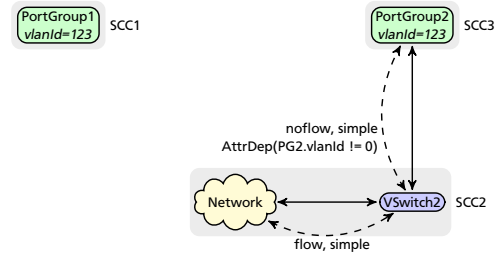


(a) System and Information Flow Models before  $\Delta_{VLAN}$ .

(b) Models after Changing PortGroup1's VLAN ID to the value 0.



(c) System and Information Flow Models before  $\Delta_{VSwitch}$ .



(d) Models after Removing VSwitch1. Breaking into three SCCs.

Figure 6.5.: Examples of Graph Deltas on the System and Information Flow Models.

us notifications on reachability changes. The SCC component graph  $G'_C = (V'_C, E'_C)$  is derived from  $G_I$ , and a new  $G_C = (V_C, E_C)$  from  $G_I$ . We compute the removed SCCs  $V'_C \setminus V_C$  and removed inter-SCC edges  $E'_C \setminus E_C$ . As well as new SCCs  $V_C \setminus V'_C$  and new inter-SCC edges  $E_C \setminus E'_C$ . We process the connectivity changes as reflected in the component graph changes in the following ways.

- **Removed SCCs or inter-SCC edges:** In the case of *reduced* connectivity, we find all iedges that have a connectivity dependency *with* a connectivity path. These iedges depend on a found path and the reduced connectivity may have invalidated this path. In particular we have to check the iedges with a path that contains a removed SCC or inter-SCC edge if their connectivity dependencies are still valid.
- **New SCCs or inter-SCC edges:** In the case of *increased* connectivity, we find all iedges that have a connectivity dependency *without* a connectivity path. In contrast to the previous case, these iedges exists because no path was found for their connected statements. The increase in connectivity may have established such a missing path. We have to check the connectivity dependency of all iedges without a connectivity path.

In our case study, if the Network node is removed from the example of Figure 6.4, SCC2 splits into two new SCCs, i.e., SCC2 is removed and two new SCCs are added. In the example, the connectivity-dependent edge between the port groups is affected and removed, because SCC2 appears in its connectivity path. Since another connectivity path could exists for the connectivity endpoints, we re-evaluate the complex rules for the removed edges' node pairs. However, in the example no other connectivity path exists.

### 6.4.3 Algorithm Analysis

In this section we discuss the termination and complexity of the dynamic analysis algorithm. Further we show the equivalence between the fully and differential information flow analyses.

---

### 6.4.3.1 Algorithm Termination and Complexity

---

The termination of the dynamic information flow analysis is given by the same set of properties as for the static analysis (cf. Section 6.3.5.1):

- *Finite Sets*: The differential analysis operates on the finite sets of the graph delta with created/erased vertices and edges, as well as attribute modifiers. The iteration over these sets processes each element only once and does not modify the sets. In particular our definition of graph deltas only allow deletion and modifications of existing nodes and not of newly created ones in the same delta.
- *Limited Inter-Rule Dependencies*: Although attribute and connectivity changes may result in the invalidation and re-evaluation of iedges, the termination of the algorithm is given by the following properties. Attributes are only changed through system model changes and not through the application of rules. Therefore one rule is not attribute-dependent on another rule, thus no inter-rule attribute dependencies exist. Complex rules can be connectivity-dependent, but we limit this dependency to connectivity produced only by simple rules. Simple rules cannot be connectivity-dependent. This one-way dependency (cf. Section 6.3.5.1) prevents any circular dependencies among complex rules.

We analyze the complexity of our differential algorithm with regard to the processing of system model changes and the handling of connectivity changes. The complexity of processing the system model changes is the following:

- *Attribute Changes*: Processing of attribute changes is linear to the number of modifiers  $|M|$ . For each modifier  $(v, a, d)$  we perform a constant lookup of the affected attribute dependencies for  $(v, a)$ .
- *Eraser Edges*: The removal of iedges is linear to the number of removed system model edges  $|E^-|$ .
- *Eraser Nodes*: For each removed node in  $|V^-|$  we remove the incoming and outgoing iedges, which is linear to the number of iedges with a constant lookup of connectivity-dependent iedges indexed by endpoints.
- *Creator Nodes*: Given the node set  $V'_S$  of the previous system model, we need to evaluate the complex rules for the pairs between the existing nodes and the new nodes:  $V^+ \times V'_S + V'_S \times V^+$ , as well as between the new nodes themselves  $V^+ \times V^+$ . For each we perform the rule application (*EvalRule* and *ImplicitDeps*) and an edge insertion.
- *Creator Edges*: We evaluate the simple rules for all new edges:  $|E^+| \cdot |R_{simple}|$ , for each edge we evaluate the rule and find the implicit dependencies. We further find the connectivity-dependent iedges where the edge establishes the relation between a node and endpoint.

The complexity of the connectivity change processing is the following:

- *Reduced Connectivity*: We are iterating over the connectivity-dependent iedges that have a connectivity path (subset or equal of  $E_I$ ). An iedge is affected if an element of its connectivity path is removed. For an affected edge we try to find an alternative path (linear to the size of the component graph, BFS for shortest path).
- *Increased Connectivity*: We are iterating over the connectivity-dependent iedges that have no connectivity path (subset or equal of  $E_I$ ). For all iedges we need to check if a path has been established between the connectivity endpoints (linear to the size of the component graph, BFS for shortest path).

---

## Complexity Comparison with Traversal Analysis

The analysis approach of Chapter 3 is not designed to handle a dynamic system model. In particular rules dependent on the current color or color tag lead to an information flow state that highly depends on the current system model. Changes to the system model requires to re-run the entire analysis. Therefore the comparison boils down to the differential complexity as previously discussed and the full analysis complexity of Chapter 3 as discussed in Section 6.3.5.1.

---

### 6.4.3.2 Full and Differential Analyses Equivalence

---

The objective is that there is no difference in the information flow graphs produced by the differential analysis compared to the full one. Given the current system model graph  $G_S$ , a system model change  $\Delta$ , and the information flow graph  $G'_I$  of the previous system model graph  $G'_S$ . The full information flow analysis (cf. Section 6.3) of  $G_S$  produces  $G_{I,\text{full}}$ . The differential information flow analysis using  $G'_I$  and  $\Delta$  produces an information flow graph  $G_{I,\text{diff}}$ . Both  $G_{I,\text{full}}$  and  $G_{I,\text{diff}}$  are equal, i.e., the edge sets are equal and all edges have the same flow type. We show the equivalence between the full analysis on the changed system model and the differential analysis based on the graph delta in the following cases:

- **Node Attribute Changes:** The full analysis would never have seen the original vertex attributes, only the changed attribute value. The regular rules application of attribute-conditioned rules may either match in a first matching semantic, or no match.

For the differential analysis we have to consider two cases: would the same rule that was applied still hold, i.e., is the attribute condition fulfilled, and would a previous rule match instead. To achieve the same results as the full analysis, the differential analysis needs to handle the two cases: the rule does not match anymore (attribute condition not fulfilled), that means the iedge has to be removed. Second, a previous rule now matches (first matching semantic), therefore the iedge produced by the current rule has to be removed. We achieve this through attribute dependencies and implicit dependencies.

- **Eraser Nodes and Edges:** The full analysis would never create iedges to or from any erased node (complex rules) nor based on any removed edges (simple rules). The differential analysis achieves the same result by removing the iedges that connect to any removed node and the iedges corresponding to the removed edges.

In addition, the removal of edges can also break the relation between nodes and their connectivity endpoints. The differential analysis removes the complex iedges where the relation is broken due to edge removal.

- **Creator Nodes and Edges:** The changed system model is given as  $V_S = (V'_S \setminus V^-) \cup V^+$  and  $E_S = (E'_S \setminus E^-) \cup E^+$ . We already showed the equivalence for the eraser nodes and edges, therefore we now consider  $V_S = V''_S \cup V^+$  and  $E_S = E''_S \cup E^+$  with the erasers already applied in  $V''_S$  and  $E''_S$ .

Given the new edge set  $E_S = E''_S \cup E^+$ , for the full analysis we can split the rule application (cf. algorithm 1) into iterating over  $E''_S$  and iterating over  $E^+$ . The differential analysis already iterated over  $E''_S$  for the previous model and now only iterates over  $E^+$ .

Similarly for the new vertex set  $V_S = V''_S \cup V^+$ . The full analysis will evaluate the complex rules on typed node pairs of  $V_S$ . We can split the evaluation into the set of typed node pairs of  $V''_S \times V''_S$ , as well as evaluating the sets  $V^+ \times V''_S$ ,  $V''_S \times V^+$ , and  $V^+ \times V^+$ . The differential analysis already evaluated the node pairs of  $V''_S \times V''_S$  for the previous model, and now only considers the pairs between the new vertices and the existing ones.

Additionally, the creation of new edges can establish the relation between vertices and their connectivity endpoints in case of (negative) connectivity-dependent iedges. We handle this case in the creator edge processing and re-evaluate the affected iedges.

---

## 6.4.4 Summary

---

Building upon the concepts of the static information flow analysis of Section 6.3 we defined a fully dynamic analysis. The key concepts are:

- *System Model Changes as Graph Deltas*: Changes to the system model, which is a graph model, are defined as graph deltas consisting of creator nodes and edges, node modifiers, and eraser nodes and edges.
- *Processing of System Model Changes*: Given a system model and a system model change as a graph delta, the differential analysis computes the information flow model changes based on the graph delta. We show that the full and differential analyses result in the same information flow model.
- *Processing Information Flow Changes*: Changes to the system model result in changes in the information flow model. The differential analysis processes how the connectivity changes and determines the connectivity-dependent edges that are affected.

Overall the differential analysis builds upon the full analysis and partially applies the rule evaluation. The attribute and connectivity dependencies of edges are essential in producing an equal information flow result for the differential analysis compared to the full one.

---

## 6.5 Implementation

---

We implemented the differential information flow algorithms in Scala, a functional as well as object-oriented programming language. In particular we want to highlight the ability of using Scala to express the information flow rules. Scala is an object-oriented language and thereby provides sub-typing, which is required to represent our hierarchically-typed vertex model (cf. Definition 35).

Further, the language provides *pattern matching* that allows to express a set of *cases* given as variables and types, which are matched against an input object and the case that matches the input object's types returns a value. In Listing 6.1 we encoded the rules 1, 2, and 3 from Table 6.1 using a pattern match. Each rule is represented by one case with a tuple of variables and optionally their types. The input object for the pattern match is a tuple of source and destination vertex representing an edge in the graph model. In our example, the first two cases require the vertices to be of type `VSwitch` and `VMwarePortgroup`, whereas the last case does not impose any type conditions on the tuple values.

```
val simpleRules: Rules.SimpleRulesMatcher[RealNode] = {
  case (vs: VSwitch, pg: VMwarePortgroup) if pg.vlanId != 0 =>
    InfoFlowEdge(vs, pg, AttrNoFlow(pg, "vlanId"), () => pg.vlanId != 0)
  case (pg: VMwarePortgroup, vs: VSwitch) if pg.vlanId != 0 =>
    InfoFlowEdge(pg, vs, AttrNoFlow(pg, "vlanId"), () => pg.vlanId != 0)

  // Default flow rule
  case (x, y) => InfoFlowEdge(x, y, Flow)
}
```

**Listing 6.1:** Information flow rules encoded as a pattern match in Scala.

Scala also allows to further restrict each case with conditions, for instance on the matching variables' attributes. In our example rules we restrict that the matched `VMwarePortgroup` named `pg` has a `vlanId` of not 0. The last rule does not have neither type restrictions nor conditions, i.e., it can match any tuple.

The evaluation of the pattern match cases follows a first matching order, thus the last rule only matches any tuples that have not been matched by a previous case. The Scala compiler provides checks on the validity of the pattern match. It will check for exhaustive matching, i.e., if all cases of possible input types are caught by at least one pattern match case. Further the compiler checks for type-based override, i.e.,

---

one case is not reachable because a super-typed case overrides it. To some degree also condition-based override checks are performed, e.g., it checks if a case with a condition follows a case with the same type but without any conditions.

The result of a matching case is an `InfoFlowEdge` that contains the source and destination vertices, the flow decision, and predicates as anonymous functions determining the validity of the edge for the attribute and connectivity dependencies. We have multiple classes for representing the flow decision, including for instance flow/noflow decisions that are attribute dependent (`AttrFlow` and `AttrNoFlow`) and which contains the vertex and attribute name of the dependency.

The rules are written in Scala and dynamically loaded using Scala's run-time reflection. Thereby the information flow analysis does not have to be recompiled when the rules change. Still the rules are statically type checked and compiled by the Scala compiler. We use the *ScalaGraph* library to represent the system model as a graph using a customized edge class (`InfoFlowEdge`). On this graph model we compute the components graph using a slightly optimized SCC compute algorithm from *ScalaGraph*.

The first implementation using Scala's pattern match approach has a set of limitations compared to the designed algorithms. First, the condition ordering respectively the condition-based override check by the Scala compiler is very simple and does not incorporate our attribute and connectivity based ordering (cf. Section 6.3.3.2). Second, we cannot control the evaluation of the patterns nor are we notified when a case has not been matched, thus we are not able to build up the *implicit dependencies*. In the first implementation we have to handle the implicit dependencies explicitly by introducing extra rules, for instance, as shown in Listing 6.2 which have to be placed before the default flow rule. Although the first implementation is not as generic and powerful as the designed algorithms, it was able to handle the rules and analysis of our case study of Section 6.2.

```
case (vs: VSwitch, pg: VMwarePortgroup) =>
  InfoFlowEdge(vs, pg, AttrFlow(pg, "vlanId"), () => pg.vlanId == 0)
case (pg: VMwarePortgroup, vs: VSwitch) =>
  InfoFlowEdge(pg, vs, AttrFlow(pg, "vlanId"), () => pg.vlanId == 0)
```

**Listing 6.2:** Explicit rules to handle the implicit dependencies before the default rule.

In terms of testing, we perform system model changes to a small and known system model graph where we specify the expected connectivity after the change as test case assertions. This allows us to test the expected changes in the information flow graph performed by the information flow rules upon system model changes. Further, we test the entire analysis on randomized system models as input data, for which we developed a system model generator using *ScalaCheck*. This allows us to test for unexpected termination issues or exceptions on unknown system models, but does not test the expected connectivity in those system models. We cover the following cases for our portgroup related rules.

- Check initial information flow connectivity for VMs on portgroups with the same VLAN identifiers, which is our known base case.
- Changing a portgroup's VLAN ID to 0. This must trigger the implicit dependency rule and the attribute dependency invalidation due to an attribute change.
- Changing a portgroup's VLAN ID to an existing one. This must trigger the regular portgroup rule due to an attribute change.
- Disconnect the vswitch of a portgroup. This must invalidate the default flow edge due to edge deletion as well as the connectivity-dependent complex edge.
- Connect a VM through a new VNIC. This establishes a new default flow through these node and edge additions.
- Disconnect a VM if the corresponding portgroup is deleted. This must trigger the invalidation of (simple and complex) edges due to the node deletion and connectivity path invalidation.



- 
- Create a new portgroup with an existing VLAN ID. Must trigger the creation of new simple and complex edges due to simple and complex portgroup rules.
  - Connect an existing vswitch. Triggers the creation of new simple and complex edges due to implicit connectivity dependency invalidation.

---

## 6.6 Conclusion

---

In this chapter we propose a approach for the static information flow analysis for dynamic systems. We introduce the concept of dynamic information flow graphs with user-defined flow rules. The flow rules capture the user's trust assumptions in system components and their isolation. The dynamic information flow graphs are dependent on the flow rules' conditions and changes to the system model may require re-computation of parts of the information flow graph. However compared to other approaches our analysis operates in a differential way, i.e., the analysis is updated based on the changes rather than performing an entire analysis. We apply our approach to the case study of isolation in virtualized infrastructures, where we model the infrastructure's configuration and topology as a system model graph and capture assumptions in the network isolation as flow rules. An existing system provides us with system model changes that lead to updates in our dynamic information flow graph. Security systems can build upon our information flow graph to verify isolation between system components using graph reachability in dynamic systems.

As further optimization of our approach, we propose the following directions. We aim for a graph reduction by replacing potential full-mesh graph structures, e.g., as created by default complex rules, with star topologies. For this we need to introduce the concept *information flow nodes* that can be created by flow rules. In our case study, the default complex rule for portgroups would create a noflow information flow node to connect the portgroups to. The information flow nodes are also dependent on flow rules conditions and need to be adapted based on system model changes. Furthermore, we can optimize our approach by using a dynamic reachability or SCC algorithm rather than re-computing the SCCs using Tarjan's algorithm. Finally, we aim to apply our approach to new case studies, such as for attacker propagation in digital-physical environments.



---

# 7 Near Real-Time Detection of Security Failures

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

---

## 7.1 Introduction

---

Infrastructure clouds are rapidly and dynamically changing systems due to self-service provisioning and on-demand scalability. Tenant as well as provider administrators frequently adapt the configuration of the sub-system they control, constituting in dynamic changes for the entire configuration. While configurations of multi-tenant infrastructure clouds are complex in themselves, overseeing the security consequences of many configuration changes by multiple administrators can easily be beyond the grasp of human operators.

Indeed, the configuration complexity we observe in dynamically changing infrastructure clouds calls for tool-support. Existing research in this space is mostly focused on dynamic infrastructure analysis of non-security properties [SGG12], node integrity monitoring [SSVJ13] or establishing security analyses of static systems given by a configuration snapshot (cf. Chapters 3 and 5). While the latter results give us confidence about reasoning on security consequences of infrastructure cloud topology and configurations, they suffer from blind spots due to transient security failures as well as from efficiency problems. In fact, the isolation case-study of Chapter 3, which uses the static approach, shows that the analysis of a mid-sized virtualized infrastructure required about *seven minutes* for extracting the configuration and building up a model and *one minute* on the actual analysis of the model. Performing such an analysis in a dynamic environment will lead to a backlog of changes that need to be analyzed, an increase in the response times in case of security incidents, as well as scalability and efficiency problems.

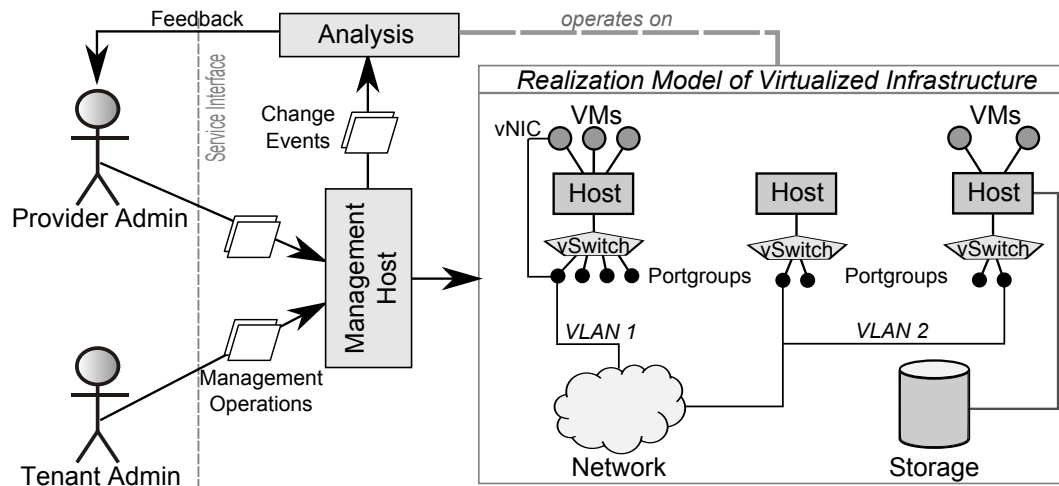
Consequently, it is our primary goal to reduce the times for configuration extraction, model building and analysis by establishing a systematic *differential* approach that does not require a full extraction and analysis on each configuration change, but still maintains strong security foundations all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

**Our Contributions.** With the overall research goal to establish a differential security analysis of dynamic infrastructure clouds, we make the following contributions: **1)** We establish an architecture that caters for near to real-time detection of configuration changes in heterogeneous virtualized infrastructures. **2)** In order to maintain a synchronized graph model of these infrastructures, we propose a set of algorithms for the computation of graph deltas (added/removed nodes and edges, changed node attributes) applicable to a graph model based on change events. **3)** We offer a practical implementation of our system, called *Cloud Radar (CR)*, for VMware environments. Our comprehensive evaluation shows that the differential approach reduces the overall analysis time significantly, putting near-to-real-time analysis in our reach.

For a broad spectrum of cloud operations and even for large infrastructures, we measure model update times in the order of milliseconds, which renders our approach several orders of magnitude more efficient than previous static analysis approaches. 4) We establish a security analysis showing that *Cloud Radar* can be set up as security monitoring of insider adversaries.

## 7.2 System and Security Model

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



**Figure 7.1.:** System model of the differential security monitoring covering compute, network, and storage.

As shown in Figure 7.1, the system model of this work is poised towards a differential analysis based on change events issued by the management hosts when the infrastructure is re-configured. The analysis system uses these change events to continuously update a graph representation of the infrastructure, the *Realization* model, which is used for subsequent analysis. As long as the management host issues the events correctly, the model covers malicious adversaries, insiders and externals alike.

### 7.2.1 System Model

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

Considering the example from Figure 7.1, we see that the *Realization* model captures all areas of the virtualized infrastructure: computing, networking and storage. While the actual model encodes fine-grained components of all these areas, e.g., storage being represented as virtual disks, file backend objects and storage pools, we focus our explanation on the networking components to prepare the ground for examples in subsequent sections. Physical hosts and their hypervisors provide networking to VMs by

---

virtual switches that connect the VMs to the network. A virtual switch contains virtual ports, to which the VMs are connected via a virtual network card (vNIC). Virtual ports are aggregated into *port groups*, which apply a common configuration to a group of virtual ports. Virtual LANs (VLANs) allow a logical separation of network traffic between VMs by assigning distinct VLAN IDs to the associated port groups. The Realization model is populated through an automated extraction of the configuration of the virtualized infrastructure from the central management host and the translation of the configuration into graph nodes and vertices. For each element in the configuration, such as a virtual machine, it constructs a corresponding model vertex and populates the required attributes. To ensure a complete translation of all relevant elements in the configuration, an element is either translated or explicitly ignored. A translation warning is thrown for elements that are not processed. Thereby we follow the same principles for discovery and translation as already discussed in Section 3.4.1, where the security impact of the complete discovery and translation has also been covered.

---

### 7.2.2 Threat Model

---

We establish a threat model based on the dependability taxonomy [ALRL04]. Agents, users, and administrators can be malicious or non-malicious. Thereby, we cover all classes of human-made faults, independent from *intent* or *capability*, that is, faults can be introduced deliberately as result of a harmful decision or without awareness; faults can be introduced by accident or by incompetence. These fault classes include misconfigurations as well as malicious insider administrators and, thereby, constitute a strong adversary model. Agents that operate on behalf of a human, e.g., due to automation, are also covered by this threat model, since we do not differentiate between the issuers of changes.

We only place one constraint on how the adversary can exert threats upon the virtualized infrastructure: The adversary is bound to the well-defined cloud manager API and cannot subvert the communication channel between the management hosts and the analysis system. In Section 7.5 we discuss and assess multiple deployment approaches to realize such a constraint in a practical environment. For example, based on isolation of the monitoring and management networks from the administrators, as well as using mandatory access control. Note that we consider the security of the software for the management host and the hypervisors as out of scope.

---

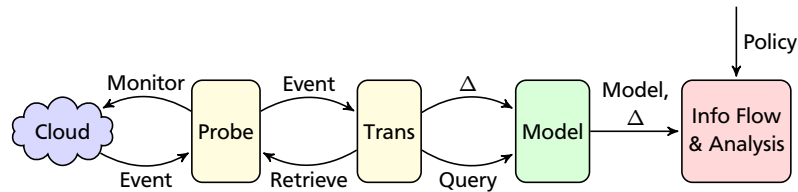
## 7.3 Design and Implementation

---

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

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

We depict the system architecture and components that implement such an analysis workflow in Figure 7.2. The system is composed of the following components. A **Probe** knows how to monitor the virtualized infrastructure and how to obtain change events. It is tied to a specific virtualization technology, e.g. VMware. For heterogeneous environments multiple probes are instantiated. For each probe a translation component (**Trans**) exists that knows how to convert from change events into model transformations, i.e.,



**Figure 7.2.:** High-Level System Workflow and Components.

a graph delta ( $\Delta$ ). Depending on the information richness of the events, further information may need to be retrieved from either the probe or queried from the existing model.

The **Model** component contains the graph model of the virtualized infrastructure as introduced in Section 7.2.1. By obtaining graph deltas ( $\Delta$ ) from the *Trans* component, the model is updated in accordance to the change event. A graph delta contains the nodes and edges that should be added or removed, as well as attribute changes for nodes. As a pre-processor for the analysis, the **Info Flow** component determines the information flow implications in the infrastructure and updates the graph model with information flow edges. In a differential analysis, the information flow is updated based on the graph delta. The **Analysis** component analyses the infrastructure given as the graph model with regard to a security policy, which expresses desired or undesired properties of the infrastructure topology or configuration.

---

### 7.3.1 Obtaining Infrastructure Change Events

---

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

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

---

#### 7.3.1.1 VMware Probe

---

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

We are using the method `WaitForUpdates` of the VMware API [VMw13a] to obtain notifications on updates and property changes for managed entities. This method is part of the *Property Collector* component, which also handles retrieval of properties of entities in the API. The method returns an `UpdateSet` object that contains an incremental version number, which is used in repeated calls to only obtain the latest changes. Further, the update contains a set of `ObjectUpdate` objects with an *Object* attribute stating the *MOR* of the updated object, as well as a *Kind* attribute to indicate the kind of update. The update kind can be i) *Enter* for a new object, ii) *Leave* for a removed object, and iii) *Modify* for property changes of that object.

Essentially, the updates state which objects have been added or removed from the inventory, and which have been modified. For new or modified objects, a set of `PropertyChanges` describe the property changes of the objects in the following form.

- **Operation:** The type of property change. It can be i) *Add* a value to a collection property, ii) *Remove* a value from a collection, or iii) *Assign* a value to the property.
- **Name:** The name of the property that is changed.
- **Value:** Only for Add and Assign, the value that is added or assigned to the property.

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

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

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

---

### 7.3.1.2 Other Discovery Probes

---

We outline the implementation of probes for other virtualization technologies and how one could obtain change events for them.

*Xen:* The XenAPI [Cit13] provides similar capabilities as VMware for obtaining change events on inventory objects. An event is composed of an monotonically increasing identifier, a timestamp, the class of the changed object (such as a virtual machine), an operation (add/delete/modify), as well as a reference and UUID referring to the changed object.

*Libvirt:* The Libvirt API [Red10] provides an event loop mechanisms for delivering change events on a VM level, such as life-cycle changes, reconfiguration etc. Events such as VM reconfiguration lack the exact configuration, therefore the probe has to obtain further information directly from the hypervisors. Further, this probe will only deliver change events on a VM level and not for other parts of a virtualized infrastructure. A combined approach with a higher-level management probe, such as the one discussed in the next paragraph, has to be pursued.

*OpenStack:* OpenStack is composed of multiple components for managing – among other aspects – compute, network, and storage resources. A central messaging service based on AMQP [Ope13] is used to exchange commands and messages between the components. By tapping into this central messaging system, we can discover what changes are requested. For example, the creation of a new VM was requested and a message is send to a compute node with the parameters of the VM. However, such events are often coarse-grained and do not reflect the changes that happen on the hypervisor-level. A combination of a high-level cloud probe with a hypervisor probe, e.g., the Libvirt probe, is desired.

---

## 7.3.2 From Change Events to Model Updates

---

From a high-level perspective, the *Trans* component translates from a change event to a model update in the form of a graph delta, as illustrated in Figure 7.2. The translation has to differentiate between three kind of change events. First, a new object appeared that may result in new nodes and edges in the graph model. Second, an object was removed and the corresponding nodes and edges in the model have to be removed, too. Finally, an object has been modified, i.e., attributes of that object have changed. This may result in attribute changes of nodes in the model too, but it can also leads to the creation or deletion of nodes and/or edges in the model. This categorization aligns well with the events produced by the VMware probe since they contain an attribute indicating the kind of change.

---

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

---

### 7.3.2.1 Translation of an Object Update

---

We explain and propose a set of algorithms for the successful translation of an Object Update into a graph delta. The handling of a failed translation due to cases such as incomplete objects or dependency ordering will be discussed in Section 7.3.2.3 and Section 7.3.2.4 respectively.

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

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

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

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

As part of the VMware translation, we distinguish between two types of objects: *Managed Entity* (ME) and *Data*. The MEs are elements in the virtualized infrastructure inventory, such as virtual machines (VM) or hosts, and have a reference (Managed Object Reference or MOR). On the other hand, each managed entity may have configuration attributes represented as *Data* objects, such as virtual devices of a VM. We represent both object types as nodes in the model.

Algorithm 2 shows the recursive translation for VMware objects. The distinction between the two object types is realized through two functions: `CreateEntity` for Managed Entities (Alg. 2), and `CreateData` for Data objects (Alg. 3). The algorithm is not shown in its entirety, but parts have been selected to illustrate its concepts. The recursive function `CreateEntity` takes the Managed Entity that needs to be translated into a model node, as well as the already created nodes and edges of this recursive translation run. If the object has already been translated, i.e., it is in the *Cache*, we simply return the cached model node. Otherwise, depending on the object's type, e.g., a virtual machine, the translation creates a new model node with the corresponding type, and extracts and translates the necessary attributes from the inventory object into the model node. An attribute can be another ME, which requires a recursive call and merging of its result, or it is a data object.

For a data object, the function `CreateData` will return the new node representing the data object. Important for the recursive translation is that we consider data objects as terminators for the recursion, because they do not link to other MEs which would lead to another recursive call. They are leaf nodes in the hierarchy graph, i.e., no outgoing edges.



---

**Algorithm 2:** Recursive Construction of Creator Node and Edge Sets. Depends on Algorithm 3 for Data Objects.

---

**Data:** New Object  $o$

**Result:** Creator Nodes  $V^+$  and Edges  $E^+$ .

CreateEntity( $o, V^+, E^+$ ) **begin**

**Input:** Object  $o$ , Creator Nodes  $V^+$  and Edges  $E^+$

**Output:** Extended Creator Nodes and Edges, Created Node

**if**  $o$  in Cache **then**

    | **return**  $V^+, E^+, \text{Cache}(o)$

**switch**  $o.type$  **do**

**case** *HostSystem* **do**

      |  $host \leftarrow \text{Host}(name = o.name, \dots)$

      |  $V^+ \leftarrow V^+ \cup host$

      | **foreach**  $vm$  in  $o.vms$  **do**

        |  $V^+, E^+, vm' \leftarrow \text{CreateEntity}(vm, V^+, E^+)$

        |  $E^+ \leftarrow E^+ \cup (host, vm')$

      | **return**  $V^+, E^+, host$

**case** *VirtualMachine* **do**

      |  $vm \leftarrow \text{VM}(name = o.name, \dots)$

      |  $V^+ \leftarrow V^+ \cup vm$

      | **foreach**  $dev$  in  $o.devs$  **do**

        |  $V^+, E^+, d \leftarrow \text{CreateData}(dev, V^+, E^+)$

        |  $E^+ \leftarrow E^+ \cup (vm, d)$

      |  $E^+ \leftarrow E^+ \cup (\text{Cache}(o.host), vm)$

      | **return**  $V^+, E^+, vm$

**case** *Network* **do**

      | /\* Ignored, because translated in host's network configuration. \*/

      | **return**  $V^+, E^+, \text{None}$

**case** ... **do**

**otherwise do**

      | /\* Unsupported object type \*/

      | **return**  $V^+, E^+, \text{None}$

  /\* Calling the recursive function \*/

**return** CreateEntity( $o, \emptyset, \emptyset$ )

---

The translation of object updates has to handle two corner cases: Incomplete objects, where the attributes of an object have not been populated fully yet, and the ordering of object updates within the same update set. We will describe the handling of these cases in Section 7.3.2.3 and Section 7.3.2.4, respectively.

### Leave Object: Deleting Nodes and Edges

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

The recursive deletion works as the following. First, given an object that was removed from the inventory, we lookup the corresponding node in the model, and add the node to the eraser node set. For all the deleted node's neighbors, we place the connecting edges in the eraser edge set. Further, we continue the recursive deletion at the neighbor node if i) the neighbor's type is a child type in the hierarchy (cf. Section 7.2.1); and ii) the neighbor node is not a managed entity.

---

**Algorithm 3:** Construction of Creator Node and Edge Sets for Data Objects.

---

```
CreateData( $d, V^+, E^+$ ) begin
  switch  $d.type$  do
    case VirtualEthernetCard do
       $vnic, vport \leftarrow VNic(\dots), VPort(\dots)$ 
       $pg \leftarrow Cache(d.backing.portgroup)$ 
       $V^+ \leftarrow V^+ \cup \{vnic, vport\}$ 
       $E^+ \leftarrow E^+ \cup \{(vnic, vport), (pg, vport)\}$ 
      return  $V^+, E^+, vnic$ 
    case ... do
    otherwise do
      /* Unsupported data type.                                     */
      return None
```

---

### Modify Object: Creating an Entire Graph Delta

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

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

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

---

### 7.3.2.2 Applying the Model Update

---

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

The graph delta has to adhere to the following invariants, otherwise the consistency of the model is not ensured. The erasers can only operate on the existing elements in  $G$ , i.e.,  $V^- \subseteq V$  and  $E^- \subseteq E$ . Elements cannot be created and deleted in the same graph delta. Further, we cannot have two node modifiers  $(n, a, v)$  and  $(n, a, v')$  where  $v \neq v'$  in the same set  $M$ , since we have a conflicting modification of the node  $n$ 's attribute  $a$ . During our evaluation we have not encountered violations of these invariants.

---

### 7.3.2.3 Postponing Incomplete Objects

---

Resolving further information of an object during the translation may fail when not all relevant attributes have been set yet. For example in VMware, when a VM is created its configuration is only later fully populated. In that case, we are dealing with an incomplete object. We maintain a set of incomplete objects and monitor updates for these objects. If an incomplete object receives an update, we try to

---

handle it as a new object rather than a modified one. If the translation succeeds, i.e., the object was complete and could be translated, we remove the object from the incomplete set. Otherwise, it remains in the incomplete object set. In practice and during our evaluation, incomplete objects always received modify events when further attributes were populated, usually within a sub-second time span. We employ periodic translation attempts for incomplete objects, in case they receive no further modify events. If an object remains incomplete until a time-out  $t_{incomplete}$  is reached, an alarm is raised that indicates the identifier of the object and when it was first observed.

---

#### 7.3.2.4 Ordering of Updates based on Dependencies

---

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

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

---

#### 7.3.3 Differential Information Flow Analysis

---

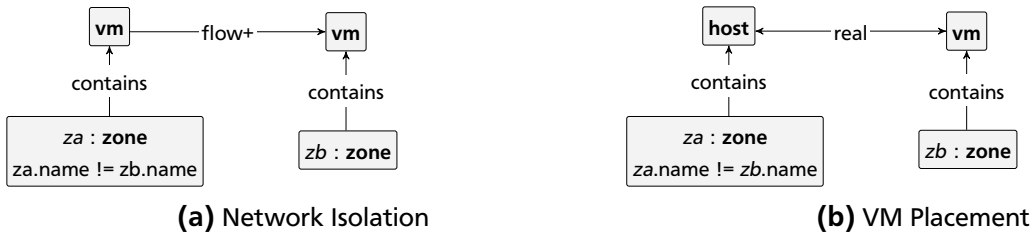
The information flow analysis determines how information may flow in the virtualized infrastructure by computing an information flow graph. The analysis operates in a dynamic way and adapts to changes in the realization model. We implement the approach of Chapter 6. The information flow graph forms the foundation for analyzing isolation properties in the infrastructure. On a high level, the analysis works in two phases: first, it takes the realization model graph that represents the infrastructure and computes an overlay directed information flow graph. Second, it computes for the information flow graph the strongly connected components (SCC), i.e., the sets of graph nodes that are mutually reachable, and constructs a reachability graph of the SCCs.

---

#### 7.3.4 Specification of Security Policies and Detection of Violations

---

For the detection of security failures, we define the following security policies, in the form of *attack states*, with their graphical representation shown in Figure 7.3. This only constitutes a subset of security policies that have been discussed in Chapter 4 and we will formalize more policies as graph matches in Chapter 8. *Cloud Radar* tries to match the policies on the dynamic realization model and information flow graph. Once a policy's attack state matches, we have found a security failure. A set of security administrators is notified about a security violation, in order to mitigate the problem.



**Figure 7.3.:** Graphical Representation of Network and Compute Security Policies.

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

The policy in Figure 7.3a operates on individual VMs, however we can make use of the aggregations such as SCCs and zones, in order to reduce the number of evaluations. Formalized in *VALID* as the following goal:

```
contains(SCC1, VM1).contains(Z1, VM1).
contains(SCC2, VM2).contains(Z2, VM2).
connected(SCC1, SCC2) & not(equal(Z1, Z2))
```

*VM Placement:* A group of virtual machines should run on one or multiple designated physical hosts, e.g., for performance, availability, or also data privacy reasons. This policy is violated if a VM runs on a different host than the ones designated. Preventing VM co-location, e.g., due to side-channel attacks [RTSS09], is a variant of this policy.

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

We have two implementations to find a policy violation by matching the policy’s attack state against the current realization model and information flow graph. The first one is a native implementation in Java/Scala that iterates through the nodes in the model graph. It benefits from a fast execution time and uses the SCC reachability graph to efficiently determine if two model nodes are connected. Either the two nodes are in the same SCC or there exists a path between their corresponding SCCs in the reachability graph. Otherwise, they are not connected. However, implementing new policies requires a native implementation, which makes it less extensible by end-users, such as a cloud administrator. The second implementation is based on a general-purpose graph matching tool called *GROOVE* [GdR<sup>+</sup>11], which tries to match a given sub-graph in a larger graph. In fact the policies in Fig. 7.3 are valid sub-graphs that *GROOVE* can match against our realization model graph. The main benefit of this approach is its extensibility, since end-users can implement new policies in a graphical and intuitive way. However, as a general purpose tool it bears a higher execution overhead and for determining connectivity it uses an equivalent but less efficient path-finding algorithm, compared to the SCC approach.

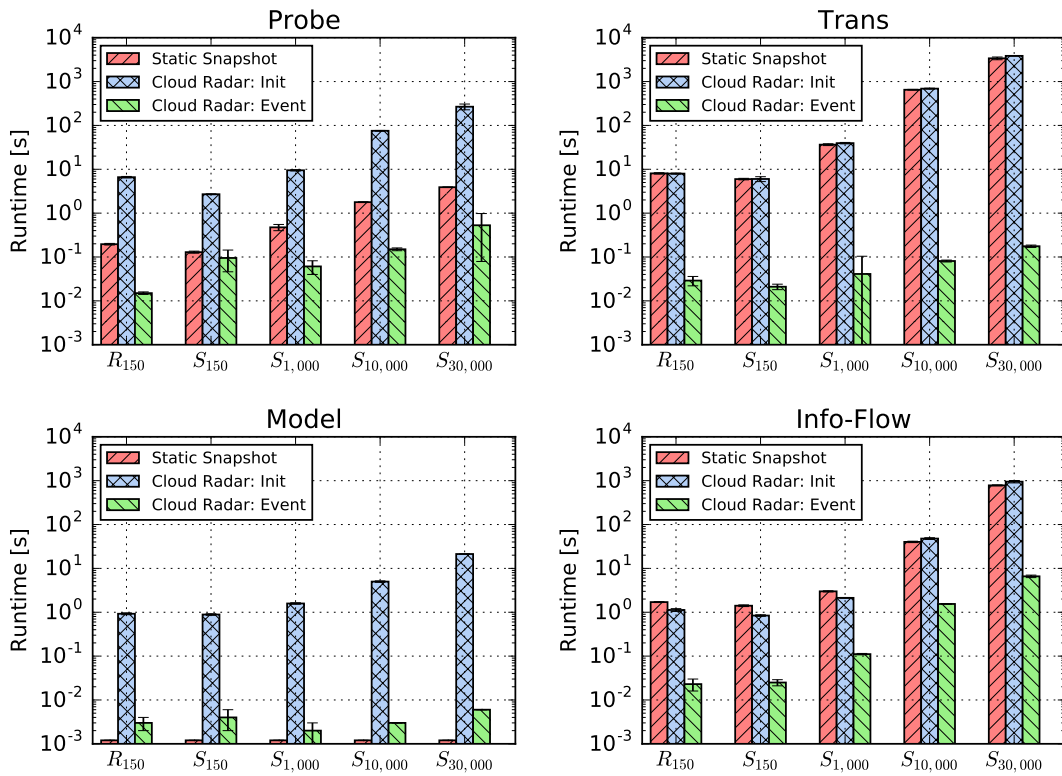
## 7.4 Performance Evaluation

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

## 7.4.1 Methodology and Environments

Our evaluation is performed with different environments: a real, semi-production environment ( $R_{150}$ ) with 2 hosts and 150 VMs, and a simulated environment that uses an infrastructure simulator incorporated in the VMware management hosts. We vary the size of the simulated environment ( $S_{\#VMs}$ ) between 150 and 30,000 VMs with a host-VM ratio of 1 : 50. This allows us to evaluate the scalability of our approach. For each measurement we perform 10 runs of the workflow. *Cloud Radar* itself runs in a Linux VM with 12 vCPUs, 12 GB RAM, and Java 1.7.

We differentiate between the two approaches of obtaining and maintaining the model of a virtualized infrastructure: *Static Snapshot* is the existing approach that always extracts the full configuration (cf. Chapter 3). In order to deal with a dynamic and constantly changing infrastructure, the extraction has to be executed periodically. On the other hand, *Cloud Radar* obtains an initial event containing the full configuration of the infrastructure, followed by events for infrastructure changes. In order to compare the performance of the two different modes, we measure the runtime of the Probe, the time needed to translate the Probe output into a model (sub)-graph, to initialize or update the graph model, as well as to construct or maintain the information flow graph. For the new event-based approach, we measure these aspects for both the initial event and subsequent change events. In order to trigger change events, we automatically perform a variety of operations on the virtualized infrastructure. In the particular measurement of Fig. 7.4 we used the *CreateVM* operation.



**Figure 7.4.:** Runtime measurements (in seconds, log scale) of the existing approach “*Static Snapshot*” and the new *Cloud Radar* approach (“*Init*” for initialisation and “*Event*” for change events) for the four system components in relation to the infrastructure size (number of VMs).

## 7.4.2 Results and Discussion

Figure 7.4 illustrates the main results of our performance evaluation with the runtime in seconds on the logarithmic y-axis, and the different environments and sizes in terms of number of VMs on the x-axis.

We break down the results into measurements for the four system components. The measurements for the existing approach are shown as the *Static Snapshot* bars, and the new approach is broken down into measurements for initialization (*Cloud Radar: Init*) and events (*Cloud Radar: Event*). Our measurement resolution is *1ms* and measurements such as the runtime of the probe for events is equal or below that resolution. Each measurement result is computed as the mean of 10 runs and the standard deviation is shown as error bars.

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

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

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

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

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

**Table 7.1.:** Breakdown of Info Flow Runtime (in *ms*) into Simple/Complex Traversal Rule and SCC Computation for Init- and Event-based Approaches.

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

**Info Flow:** If we break down the information flow analysis (cf. Table 7.1), we observe for the full analysis a linear scalable evaluation of simple traversal rules and SCC computation. We also see a quadratic complexity for evaluating the complex traversal rule, which needs to evaluate pairs of port groups in

our example rule set. The differential approach provides significant improvements for the event case of Table 7.1 with a rules evaluation that depends on the size of the event and a linear SCC computation. We discussed potential optimizations for reducing the quadratic complexity of complex rules and SCC re-computations in Chapter 6.

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

In summary and in the light of the more expensive initialization of the new approach, when does it actually pay off? We define the break-even of the event-based analysis compared to the static analysis as the following. Given the cumulative timings for static analysis  $t_s$ , CR initial analysis  $t_i$ , and event-based analysis  $t_e$ . We define the break-even of the event-based analysis compared to the static analysis for the number of events  $x$  as  $\frac{t_i + x t_e}{t_s + x t_s} \leq 1$ , where  $t_i$  and  $t_s$  are required for initialization and followed by  $x$  recurring event processing (either  $t_e$  or again  $t_s$  for static analysis). Thus,  $x \geq \frac{t_i - t_s}{t_s - t_e}$  and for a discrete number of events follows  $x = \left\lceil \frac{t_i - t_s}{t_s - t_e} \right\rceil$ . Table 7.2 shows the cumulative timings and the break-even ratio  $\frac{t_i - t_s}{t_s - t_e}$ . For instance, consider the 10,000 VM environment  $S_{10,000}$  and the cumulative runtimes of both approaches. The initialization in the existing approach overall takes  $693s \pm 12$  and for the new approach  $819s \pm 16$ . While follow up model updates require a full periodic execution of the entire workflow in the existing approach, the new one only requires 1.8s for each change event. Although the new approach is slightly more expensive in the initialization, even with just one event after initialization it pays off due to the much more efficient event processing, which is also true for all other environments.

**Table 7.2.:** Cumulative analysis timings (in ms) and number of events for break-even.

	Static	CR: Init	CR: Event	Break-even ratio
$R_{150}$	$9968 \pm 291$	$16627 \pm 345$	$70 \pm 10$	$0.67 \pm 0.05$
$S_{150}$	$7515 \pm 246$	$10435 \pm 771$	$145 \pm 49$	$0.40 \pm 0.11$
$S_{1000}$	$39931 \pm 1753$	$52669 \pm 1490$	$215 \pm 66$	$0.32 \pm 0.06$
$S_{10000}$	$693001 \pm 12487$	$818554 \pm 16340$	$1778 \pm 19$	$0.18 \pm 0.03$
$S_{30000}$	$4172772 \pm 243643$	$5092537 \pm 75140$	$7306 \pm 622$	$0.22 \pm 0.06$

*How many events can be processed per second until we run into a backlog?* Considering the simulated 10,000 VM environment ( $S_{10,000}$ ), we can observe and translate approximately 33 VM creation operations per minute, bounded by the dominating translation time of 1.8s per CreateVM operation (cf. Table 7.2 for cumulative analysis timings) and assuming a serialized processing.

## 7.5 Security Evaluation

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

### 7.5.1 Security Analysis

The security analysis considers the management host creating events and the *CR* host as separate entities and considers multiple attack vectors including manipulating network communication through VLAN re-configuration, and denial of service or dropping communication sessions in the Session Manager. We propose a practical deployment of *Cloud Radar* that is secure in the face of an insider adversary, based on

---

a small set of assumptions and a deployment pattern, which includes isolation of the reporting network, a heartbeat signal, and mandatory access control for regular administrators.

---

### 7.5.1.1 Assumptions and Deployment Pattern

---

The threat model of Section 7.2 already introduces that software attacks are out of scope, which includes that the management host software cannot be manipulated by the adversary. Our analysis is built on the following explicit assumptions, which form the basis of the deployment pattern.

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

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

The assumption [access] is motivated by vSphere Security [VMw13b] best practice, which states that hypervisor hosts should only be managed through the central management host. This can also be enforced by putting the hypervisor into *lockdown mode*, by which no other users than `vpxuser`, the vCenter management user, have authentication privileges nor can perform operations on the host directly.

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

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

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

**Authentic and Complete View of Topology and Changes:** *Cloud Radar* obtains an authentic and complete view of the topology of the virtualized infrastructure and its changes. If CR requests version  $n$  of the change events, the management host correctly produces an event  $e_n$ , which contains all changes after  $e_{n-1}$  up to reception of the request for version number  $n$ . For  $n = 0$  the management host provides an authentic and complete view of the entire infrastructure topology.



---

### 7.5.1.2 Security Argument for Event Integrity and Availability

---

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

#### Integrity

For any  $n$ -th event  $e_n$  received at *CR* holds that the event is correct, fresh, in order and as it has been sent by the management host. The management host produces an event  $e_n$ , which contains all changes after  $e_{n-1}$  up to reception of the request for version number  $n$ , which completes the event chain. We obtain the order and weak freshness properties from the version number, as an event  $e_n$  must have been generated after any event  $e_{<n}$ .

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

#### Weak Availability

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

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

The Weak Availability clause, i.e., an alarm is raised after  $t_{timeout}$  is elapsed, is obtained from the condition [heartbeat]. If the channel waits for a packet or the channel is interrupted, then we have that eventually  $t_{timeout}$  will be reached without a packet having arrived at *CR*. According to [heartbeat], the management host produces a heartbeat signal after each time window  $t_{hb} < t_{timeout}$ . Therefore, we have that the after  $t_{timeout}$  without a message, *CR* can conclude that the channel is interrupted and raise an alarm. It follows that availability failures are detected within  $t_{timeout}$ . The system will try to re-establish the connection after an interrupt, however for the duration of the re-establishment no security guarantees can be given.

---

### 7.5.2 Security Discussion

---

The integrity and availability of events is essential for a reactive and event-based security system such as *Cloud Radar*. The discovery and translation follows the methodology of Chapter 3 where everything is discovered, in our case all the change events of infrastructure elements are received, and explicitly translated or ignored.

---

As part of our event translation we may deal with objects that are temporary incomplete. In practice these incomplete objects received a completing change event within a sub-second time frame. However if it remains incomplete it can impact the detection rate of our analysis. Therefore, we raise an alarm once a time-out has been reached, and the security operator has to acknowledge the problem. Furthermore, as part of our model update we enforce invariants on the graph deltas that ensure the consistency of our model. In particular an update cannot modify the same vertex attribute with different values as part of the same change event. Further we do not allow deletion and creation of an element simultaneously in the same change event.

*Cloud Radar* uses the information flow analysis of Chapter 6, which also implies that the correctness of the analysis is based on the correctness of its inputs, in particular the information flow rules. In addition the policies and their configuration as user-defined inputs have to be correct.

---

### 7.5.3 Security Testing

---

For each policy (cf. Section 7.3.4) we determine the operation that may cause a policy violation if used with a specific parameter. We execute these operations several hundred times with a parameter from a known set of violating parameters or a complementary random parameter, similar to *Fuzzing* from software security testing. *Cloud Radar* is required to detect a policy violation in the case of a parameter from the violating set, and otherwise no violation should be detected.

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

The security testing uses the real environment described in Section 7.4.1, because the simulated one does not support all management operations and its networking configuration is not suited for the network isolation policy. This is not problematic as analysis performance and scalability are not a concern in this security testing, and a real environment yields more realistic behavior as a simulated environment.

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

---

## 7.6 Summary

---

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

---

## 8 Proactive Security Analysis of Changes

In this chapter we tackle the challenge of misconfigurations in complex and dynamic virtualized infrastructures. We establish a practical security system, called *Weatherman*, that proactively analyzes changes induced by management operations with respect to security policies. We achieve this by contributing the first formal model of cloud management operations that captures their impact on the infrastructure in the form of graph transformations. Our approach combines such a model of operations with an information flow analysis suited for isolation as well as a policy verifier for a variety of security and operational policies. Our system provides a run-time enforcement of infrastructure security policies, as well as a what-if analysis for change planning.

---

### 8.1 Introduction

---

In combating misconfigurations, insider attacks and resulting security failures, the assessment of configuration changes and rigorous enforcement of security policies is a crucial requirement. It is important to establish whether an intended configuration change will compromise the security of the system before the change is deployed. We build a practical analysis system, called *Weatherman*, that uses a model-based approach for assessing configuration changes and their impact on the security compliance of a virtualized infrastructure. We call our system *proactive* as changes are analyzed before they are deployed.

Facing an intended configuration change, *Weatherman* needs to establish first how the infrastructure would be affected by the change. Our *operations transition model* (Section 8.3.1) covers security-relevant operations and models their impact on the infrastructure configuration and topology in a graph rewriting language. For our example, it contains a model of the VMware operation `UpdatePortGroup` encoding how VLAN ID changes affect the network. Having established a what-if infrastructure model for the intended change, the next important question is: How does the information flow and isolation change in the system? *Weatherman* performs an information flow analysis in the what-if infrastructure model as an intermediary step to determine isolation properties (Section 8.3.2). Finally, the infrastructure model is checked against a variety of security and operational policies, which are implemented as graph matches and evaluated by the graph transformation engine (Section 8.3.3). Overall, our system establishes whether a future configuration change will constitute a security compromise and rejects the change if a violation is detected (Section 8.4).

In combination, the operations transition model, the what-if analysis with dynamic information flow evaluation and the subsequent graph matching with security policies realize a powerful and versatile tool. It offers a security analysis for compute, network and storage operations and policies. While we focus the initial discussions on a simple running example, we stress that a wide range of security-relevant operations have been modeled along with different styles of security policies, discussed in Section 8.3.3.

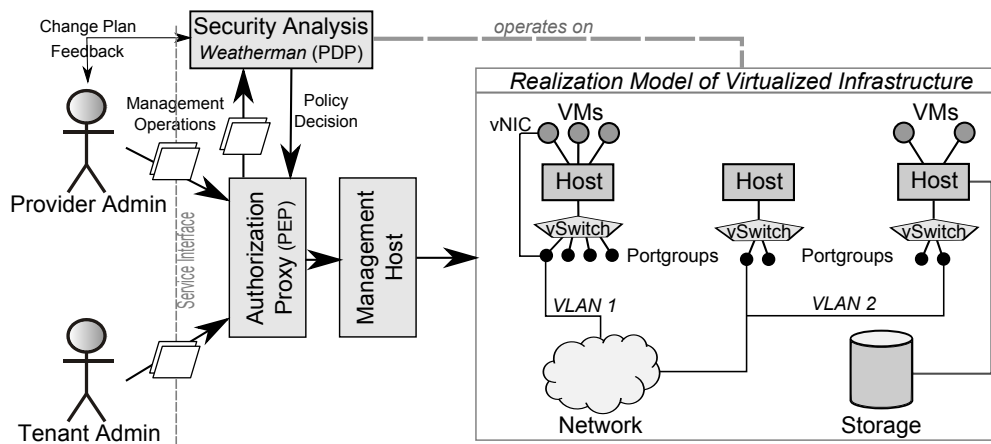
**Our contributions** are the following:

1. We propose the first formal model of cloud management operations, the *operations transition model*, that captures how such operations change the infrastructure's topology and configuration. We express the operations as transformations of a graph model of the infrastructure, which is based upon the formalism of graph transformation [Roz97].
2. We propose a unified model that integrates with the operations model the specification of security policies as well as an information flow analysis suited for isolation policies. We formalize a variety of policies, such as in the areas of isolation, dependability, and operational correctness using graph matching.

- Based on our model, we design and implement a practical security system, called *Weatherman*, which assesses and proactively mitigates misconfigurations and security failures in VMware infrastructures. We evaluate the performance of our analysis in a practical environment, and we further discuss and test the security of our system.

## 8.2 System and Security Model

In this chapter we build on the system and security model of dynamic virtualized infrastructures as presented in Chapter 7. In Figure 8.1 we illustrate our model of a virtualized infrastructure, which consists of (virtualized) computing, networking and storage resources that are configured through a well-defined management interface. We consider multiple administrators with different privileges, where the *provider* administrators govern the entire virtualized infrastructure, and *tenant* administrators manage an assigned logical resource pool.



**Figure 8.1.:** The System Model contains a topology model of the virtualized infrastructure and an authorization proxy acting as Policy Enforcement Point (PEP) based on the security analysis of operations by our Policy Decision Point (PDP).

The main difference is that in this work the model is poised towards a proactive analysis of operations that are intercepted at the management host. Our system consists of a Policy Enforcement Point that intercepts the management operations and denies them if they violate a security policy. The Policy Decision Point evaluates the intercepted operations and decide if they would cause a security policy violation if deployed to the current infrastructure. Furthermore, we allow administrators to submit change plans directly to the security analysis system in order to determine before hand if they may cause a security violation.

## 8.3 A Model of Dynamic Virtualized Infrastructures

We capture multiple aspects relevant for the analysis and integrate them into a unified model based on graphs and graph transformations. We represent the topology and configuration of the virtualized infrastructure, establish how the infrastructure can be changed by management operations, and verify the infrastructure with regard to security policies. As we are focusing on isolation properties, we further need to determine information flows in the system.

### 8.3.1 Modeling of Infrastructure Changes

We model the impact of management operations in terms of infrastructure changes using graph transformations. We will briefly introduce the formalism and describe our methodology how we can create a model for a practical system, followed by concrete examples of models for specific VMware operations.

---

### 8.3.1.1 Modeling Operations as Graph Transformations

---

We model each operation as a graph transformation rule, which takes the graph representation of the virtualized infrastructure as input and transforms it into a modified one. According to [Roz97], we define a graph transformation rule as the following.

**Definition 40** (Graph Transformation Rule). *A graph transformation rule  $p$ , also called a production rule, has the form  $p : L \rightarrow R$ , where graphs  $L$  and  $R$  are denoted the left hand side (LHS) and right hand side (RHS), respectively. The production morphism  $r$  establishes a partial correspondence between elements in the LHS and the RHS of a production, which determines the nodes and edges that have to be preserved, deleted, or created. A match  $m$  finds an occurrence of  $L$  in a given graph  $G$ , then  $G \xRightarrow{p,m} H$  is an application of a production  $p$ , where  $H$  is a derived graph.  $H$  is obtained by replacing the occurrence of  $L$  in  $G$  with  $R$ .*

An important extension to graph transformations are application conditions that express constraints on the applicability of a production rule, which includes constraints on the attribute values of vertices. Further, parametrized rules capture expected attribute values as parameters that need to be satisfied by the application condition. This is important for our model as management operations are parametrized. The *Operations Transition Model* consists of multiple graph transformation rules and captures how management operations change the topology and configuration of a virtualized infrastructure.

**Definition 41** (Operations Transition Model). *The Operations Transition Model consists of named and attribute-parametrized graph production rules which are specified as the set  $P = \{p_1, \dots, p_n\}$ . Each rule corresponds to a parametrized management operation  $op$  and models the effects of  $op$  on the infrastructure as graph modifications on the Realization graph model. The name of each production rule corresponds to the name of the management operation.*

The ordering of the rules is not relevant for the modeling as the rules model the operations independently. However, the ordering becomes important for the analysis (cf. Section 8.4) which performs an ordered application of a subset of rules with parameter values on a given infrastructure model graph.

---

### 8.3.1.2 Modeling Methodology

---

For any existing real-world virtualized infrastructure like VMware, the API documentation does not offer a precise formal definition and model, but rather a semi-formal description of the operations. A contribution of this work is to create a formal model that allows for precise statements to be made and proved or refuted. It is of course not possible to formally prove that our formal model captures the informal description, however there is a methodology to obtain a “good” model by combining the following directions:

**1) API Documentation:** We follow the API documentation that describes for each operation the functionality, the required parameters as well as the preconditions and effects that the operation has on the infrastructure. For the relevant operations, we determine the parameters that are security-critical and which will have an impact on the model when the operation is performed. Overall, the API documentation provides us with a list of relevant operations, their parameters, and a high-level idea of their impact on the infrastructure.

**2) Infrastructure Change Assessment:** In order to understand how the infrastructure is changed in detail by an operation, we inspect the configuration of the infrastructure *before* and *after* the operation has been issued. For each operation that we have selected based on the API documentation, we vary the parameter values to determine their different effects, if applicable. For example, varying the VLAN identifier parameter of a virtual network re-configuration preserves the same effect, whereas varying the device configuration of a new virtual machine creation may lead to different topology changes, e.g., attaching the VM to a different virtual network. We do not only study the differences in the configuration

**Table 8.1.: Overview of Security-Critical VMware Operations [VMw11].**

Operation	Description	Policy Impact
AddPortGroup	Creates a new port group on a given host and virtual switch, with a name and VLAN ID.	Network Isolation
UpdatePortGroup	Updates the name and/or VLAN ID of an existing port group on a given host.	Network Isolation
RemovePortGroup	Removes an existing port group on a host given by name.	Dependability
UpdateNetworkConfig	Updates the network configuration of a host; another means of creating or updating port groups.	Network Isolation
CreateVM	Creates a VM on a host with virtual storage and network resources (modeled as sub-operations).	Compute Placement
AddVirtualDisk	Creates a virtual disk for a VM with file backend.	Storage Isolation
AddVirtualNic	Creates a virtual NIC connected to a port group.	Network Isolation
ReconfigVM	Updates a VM's configuration, including storage and network resources.	
UpdateVirtualDisk	Updates the file backend of a virtual disk.	Storage Isolation
UpdateVirtualNic	Connect a virtual NIC to a new port group.	Network Isolation

after each operation, we also investigate the differences in the resulting graph models. The changes from the graph model of the configuration before the operation was performed and the graph model after the operation guides us how a graph production rule of the operation may look like. The graph model changes include new and deleted vertices and edges, as well as attribute changes.

**3) Validation with Administrative Tasks:** Finally, we also performed common administrative actions from the graphical management client, which itself issues the documented API operations. We intercepted and analyzed these issued operations and discovered that the management client makes use of other operations from the API to perform the same task. For example, to change the VLAN identifiers of a virtual network component the usual operation is `UpdatePortGroup`, however the client software issues the much more general operation `UpdateNetworkConfig`. We extended our model to include these other variations of performing security-critical tasks.

---

### 8.3.1.3 Modeling of a Practical System

---

The VMware API (v5.0) consists of 545 methods [VMw11], but many of these operations do not affect the topology or configuration of the virtualized infrastructure, because they deal with VMware-specific management and operations aspects such as licensing and patch management, handling of administrative sessions, or diagnostics and alarms. We identified 95 operations that modify the topology or configuration of the infrastructure. We model a security-critical subset of VMware management operations as listed in Table 8.1, which also indicates potential policy violations (cf. Section 8.3.3).

Overall, this subset enables the analysis of security relevant topology and configuration changes in the areas of virtual networking, storage, and compute resources, and, thereby, the verification of isolation breaches. We consider changes to the virtual compute, network, and storage infrastructure, such as the creation of virtual machines, creation or updates of virtual switches and interfaces, and attachment of storage to virtual machines. Complex operations, such as creating VMs, are broken down into sub-operations. Although we focus on a subset of operations, the diversity of this subset in terms of resources and actions shows that our approach and methodology is generally applicable, and would extend to the remaining operations as well.

From the subset of operations, we present the production rules of two operations: The `UpdatePortGroup` operation changes the isolation property of a virtual network, as well as the sub-operation `AddVirtualDisk` of the `CreateVM` operation that connects a new virtual disk to a created VM. The two examples cover a spectrum of operation classes: First, operations that create infrastructure elements as well as updating existing ones; Second, operations that work on different resource types, namely, storage and network.

---

## Visual Notation of Graph Transformation Rules in *GROOVE*

The graph transformation rules are illustrated in Fig. 8.2. We selected *GROOVE* [GdR<sup>+</sup>11], a tool for specifying and applying transformation rules, as our graph transformation environment and the rules are shown in its visual notation. Each rule is represented by a graph that describes both the LHS and RHS (cf. Def. 40). The elements of such graph transformation rule graphs are:

**Graph Vertex** (rectangular node) represents vertices of the host graph. The type of the vertex is given as the label of the node, e.g., a host node in Fig. 8.2a.

**Attribute Node** (elliptic node) represents attribute values of a graph vertex with the label representing the value type (*string*, *int*). A superscript integer *i* on an attribute node matches the attribute value against the *i*th transformation rule parameter.

**Graph Edge** is an directed edge between two graph vertices with an edge label, for instance, *real* for system model edges or *flow* for information flow edges. If the edge label has the suffix + it represents a non-empty path of such edges.

**Attribute Edge** is an edge between a graph vertex and an attribute node. The edge label represents the attribute name.

**Quantifier Nodes and Edges** (dotted line) allow quantification in rules over sub-graphs. The quantifier is represented as a quantifier node (visually: rectangular, dotted) and quantifier edges labeled with @ connect the applicable sub-graph to the quantifier. Both universal as well as existential quantification can be formalized. Quantifiers can be nested using quantifier edges labeled with *in*.

Furthermore, the shapes of graph vertices and edges as well as attribute edges capture the application conditions and modifications of a graph transformation rule.

**Readers** (thin line) are nodes and edges that need to be matched in the graph for the rule to be applicable and which are preserved in the transformation, i.e., they belong to both the LHS and RHS. For example, the host node in Fig. 8.2a is a reader node. Reader elements form a *positive* application condition of the rule, i.e., the rule only matches if the application condition is fulfilled.

**Creators** (bold line) capture nodes and edges that are created and which only belong to the RHS. For instance, the *vdisk* node and its associated edges in Fig. 8.2a are creator elements. These elements are added to the resulting graph as part of a matching rule.

**Erasers** (thin dashed) are nodes and edges that need to be matched, and which will be deleted by the transformation, i.e., only belong to the LHS. In Fig. 8.2b attribute edges are deleted as part of the rule.

**Embargoes** (thick dashed) are nodes and edges that need to be *absent* in the graph, in order that the rule matches. They form a *negative* application condition. For our policy formalization in Section 8.3.3 we use embargoes extensively.

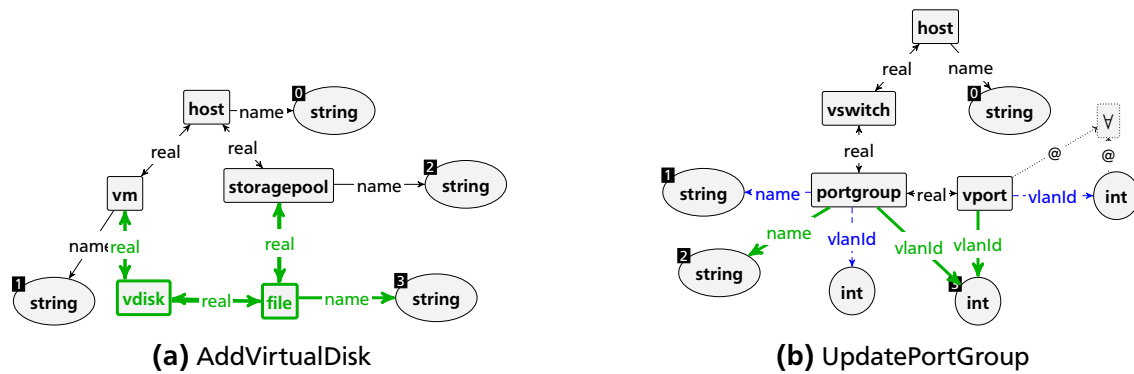
**Conditional creators** (thick dashed and bold line) combine creators with embargo elements. Such creator elements are *conditionally* created only if the elements are not yet present in the graph. We use conditional creators for our formalization of information flow rules in Section 8.3.2.

For further introduction and illustrative examples of the *GROOVE* visual notation we refer to Ghamarian et al. [GdR<sup>+</sup>11].

### Disk Creation Operation: `AddVirtualDisk`

```
AddVirtualDisk(string hostname, string vmName, string storagepool, string filename)
```

As part of the creation of a virtual machine, a virtual disk is created and attached to the VM, which is identified by a given hostname and the VM name. Virtual disks are file-based (given by a filename), and the file is residing on a storage pool, given by a name. The production rule of Fig. 8.2a finds the



**Figure 8.2.:** Examples of storage and network operations as modeled as graph transformation rules in *GROOVE*.

corresponding sub-graph where the names of host, VM, and storage pool match the rule’s parameters. New nodes for the virtual disk (vdisk) as well as the file backend (file) are created and connected to the matched sub-graph by specifying them as creator elements (visually: thick, green). In *GROOVE*, attributes of a node are represented by data nodes, visually indicated as ecliptic shapes, that are connected by a labeled edge, where the label denotes the attribute name. The numeric superscript on data nodes show that an attribute value is matched against a rule parameter, e.g., the host’s name is matched against parameter 0.

### Virtual Network Update Operation: UpdatePortGroup

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

Using this operation, an administrator can change the configuration of an existing port group. The port group is identified by its name, as well as the host where it resides on, and the operation allows to change the port group’s name and VLAN ID. In the rule of Fig. 8.2b, changing attributes is modeled as changing the edges to different data nodes based on the input parameters. The VLAN ID is not only contained in the port group nodes, but also in the associated vport nodes, i.e., virtual switch ports. Therefore, changing the VLAN ID of the port group also requires to change the VLAN ID of all virtual ports associated to that port group. For this we use the universal quantifier  $\forall$  that applies a sub-rule, given by nodes connected to the quantifier with @ labeled edges, to all its matches [RK09]. In this case, it updates the vlanId attributes of all matching vport nodes.

#### 8.3.1.4 Atomicity of Changes

A crucial point is the use of the universal quantifier in the graph production rules. This allows us to formalize an operation like `UpdatePortGroup` as an *atomic* action: the action changes several nodes in the network simultaneously. Observe that we cannot give a bound on how many nodes will be affected, and the universal quantifier allows us to change all affected nodes in a single transition. This is both relevant for modeling and for efficiency of the verification, and in fact a classical problem [LS89, Lip75, Lam90]. For what concerns modeling, our atomic actions are in general *not* equivalent to a model with “small” transitions that involve changing only one node at a time for example. To see that, suppose administrators accidentally issue two conflicting `UpdatePortGroup` operations. In a model with “small” transitions, the interleaved execution of the changes may then lead to an inconsistent state that is not reachable by any “big” atomic `UpdatePortGroup` transition. In a way this model forbids the “parallel” execution of conflicting updates and only allows what is equivalent to their serialization.



---

### 8.3.1.5 Operations Modeling Summary

---

We use graph production rules to model the effects on the configuration and topology of a virtualized infrastructure for a variety of security-critical management operations. We propose a methodology how such a model can be established for a practical virtualization system, such as VMware. The methodology builds on an informal API description, comparing the graph models before and after an operation has been applied, as well as validating the expected changes using common administrative tasks.

---

### 8.3.2 Dynamic Information Flow Analysis

---

Our information flow analysis computes potential information flows within the infrastructure, and thus enables the system to determine isolation failures between tenants. Our operations model lead to a dynamic system model and therefore requires a dynamic information flow analysis.

The information flow analysis of Chapter 3 operates on a static system model and uses a graph coloring and traversal approach based on a set of traversal rules. The traversal rules define for a pair of connected Realization model vertex types if the traversal and coloring should proceed or not. The rules further consider the traversal direction, vertex attributes, and the current graph color. We adapted the existing traversal rules, which capture best-practices on virtualization and network security, and formalized them as graph production rules. However, the challenge of such a formalization is that a direct encoding of the graph coloring approach in *GROOVE* would result in an expensive blow-up of the state space. Therefore we opted for the construction of an information flow graph instead of performing a graph coloring. We formalize the approach of dynamic information flow graphs (cf. Chapter 6) as graph transformations in *GROOVE*, in order to integrate it in our overall analysis approach. A set of graph production rules capture trust assumptions on the isolation of particular infrastructure elements, and construct the information flow graph by introducing edges that denote if flow is either permitted or denied.

This formalization has multiple advantages. First, many of the original graph traversal rules decide graph traversal, i.e., information flow, for a pair of vertices independent of the current node color. We can greedily introduce information flow edges between all such vertices without causing a large state space, leveraging *GROOVE*'s universal quantifiers [RK09]. Second, we have a direct mapping between edges and conditions in the realization graph and the constructed information flow edges. This allows us to react on changes in the realization model and determine their effect on the information flow graph. Finally, we can determine connectivity between any pair of vertices after the information flow graph has been computed, instead of performing a new graph coloring from each information sink. In particular in combination with the computation of strongly connected components on the information flow graph, we can efficiently determine such connectivity.

---

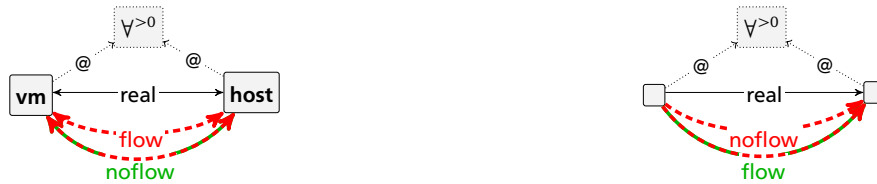
#### 8.3.2.1 Information Flow Rules and Application

---

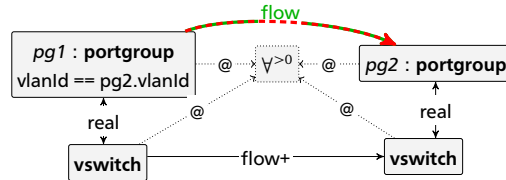
We differentiate between three kind of information flow rules: A *simple* rule describes information flow between a pair of adjacent vertices given by their types with potential conditions on the vertices' attributes. A *default* rule is a simple rule that matches any pair of adjacent vertices without any conditions. Finally, a *complex* rule describes information flow between non-adjacent vertices.

##### Simple Information Flow Rules

The first kind of rules are used both when the information flow is computed for the first time on the initial graph, or when new edges are added. They are simple in the sense that they work on directly adjacent nodes connected by a Realization model edge (*real*), and either introduce a directed information flow edge for *flow* or *noflow*. Fig. 8.3a shows a simple information flow rule that stops information flow between a host and a virtual machine (*vm*) by creating bidirectional *noflow* edges between them,



(a) Simple Stop Rule: Trusted hypervisor prevents flow from VMs to Host and vice versa. (b) Default Rule: Flow between all pairs of connected vertices without noflow edge.



(c) Fast-Edge Rule: Flow between Portgroups with the same VLAN ID and connected VSwitches.

Figure 8.3.: Examples of different kinds Information Flow Rules modeled in GROOVE as Production Rules.

if not already present. This captures the (arguable) trust assumption that no side-channel information leakage exists between virtual machines on the same host [RTSS09]. The noflow edge is created with a *conditional new*, i.e., it is only created if not already present by combining a creator and embargo edge. Applying the simple rules will eventually terminate when all pairs are connected by either a flow or noflow edge. We design the rules to be *confluent*, i.e., whenever more than one explicit rule is applicable, it does not matter for the result which one we take first. We can thus use the universal quantifier  $\forall^{>0}$ , which requires at least one match for the rule to be applicable, to express that we apply the production rule to all possible matches greedily (i.e., we do not have a state explosion).

### Default Rule

The above simple rules typically represent trust assumptions on *isolation* properties of elements in the infrastructure and therefore introduce noflow edges. The flow edges are conditionally introduced by a *default* rule, as shown in Fig. 8.3b, if neither a flow nor noflow edge are present between a pair of nodes. The rule is applied when no more simple rules are applicable. Thus, the default means that we assume information may flow when the simple rules do not tell us otherwise. This may be too pessimistic, but with this over-approximation we are generally on the safe side. We achieve the operational aspect by designing simple GROOVE rule application strategies, in this case to first apply simple rules as long as possible and then apply the default rule as long as possible, i.e., until all node pairs have been evaluated.

### Complex Information Flow Rules

A direct encoding of the original graph coloring of Chapter 3 is not suitable in GROOVE as the change in the graph state leads to an expensive blow-up of the state space. A feasible alternative is the introduction of tunneling edges representing the pairs that need to have the same coloring, i.e., allowing a flow, as discussed in Chapter 6. As an example, Fig. 8.3c shows a production rule that creates a tunnel edge between two VLAN endpoints that are not necessarily directly connected by a real edge, but which are connected through a path of flow edges. Here, two VMware port groups, which are modeled as portgroup with a VLAN identifier, are hosted on different virtual switches, and the rule fires for pairs of portgroups with the same VLAN identifiers, if the underlying switches are connected. A similar rule exists when two port groups are connected to the same vswitch.

### Adjust Existing Information Flows

The dynamic information flow analysis needs to adjust the existing information flows if the Realization model graph changes. The removal of information flow edges that are connected to removed nodes is

---

covered by the underlying formalism (*Single Push-Out* [Roz97]) as dangling edges are removed. For each pair of nodes that are no longer connected by a real edge, but still feature an information flow edge, we need to remove the flow edge. This is accomplished by two production rules similar to the simple information flow rules, but with two untyped nodes, a condition that no real edge is present, and the removal of either a flow or noflow edge.

The information flow edges that are based on changed attributes are recomputed if their predicates do not hold anymore. That means, for each information flow rule that introduces an information flow edge based on an attribute condition, such as the VLAN ID attribute dependent rule of Fig. 8.3c, we have an adjusting production rule that verifies that the attribute condition still holds; if not, it revokes the information flow edge. Adjusting the information flow graph based on changes in the Realization model may further influence connectivity-dependent information flow edges, such as the ones produced by the complex portgroup rule. Similar to an adjusting production rule for attribute changes, we also have a production rule that deletes flow edges if their connectivity condition is no longer satisfied.

---

### 8.3.2.2 Information Flow Analysis Summary

---

We formalized the dynamic information flow analysis of Chapter 6 using graph transformations in *GROOVE*. We conclude with discussing the advantages and disadvantages of this formalization. *GROOVE* provides universal quantifiers that allows us to apply a rule to all matching edges in one state transition, thereby reducing the state space. Rules with a *conditional new* are only applied to non-evaluated edges which ensures the termination of our algorithm when all edges have been evaluated. The *GROOVE* control language provides a variety of statements for the application of rules, including, *alap* for applying graph transformation rules *as long as possible*, choice to select from a set of available rules, and *else* if a rule or set of rules do not match anymore.

However, *GROOVE* also bears limitations in formalizing our approach, in particular handling rule dependencies. In the generic approach of Chapter 6 we handle rule dependencies through validating the ordering of rules and inherit the conditions of mismatched child rules. In the *GROOVE* formalization we require explicit adjusting rules that check for condition invalidation and that handle the default cases. In the case-study of this work we are able to design the rules to be confluent and thereby do not have inter-dependencies, except for the default rule that handles unmatched types.

---

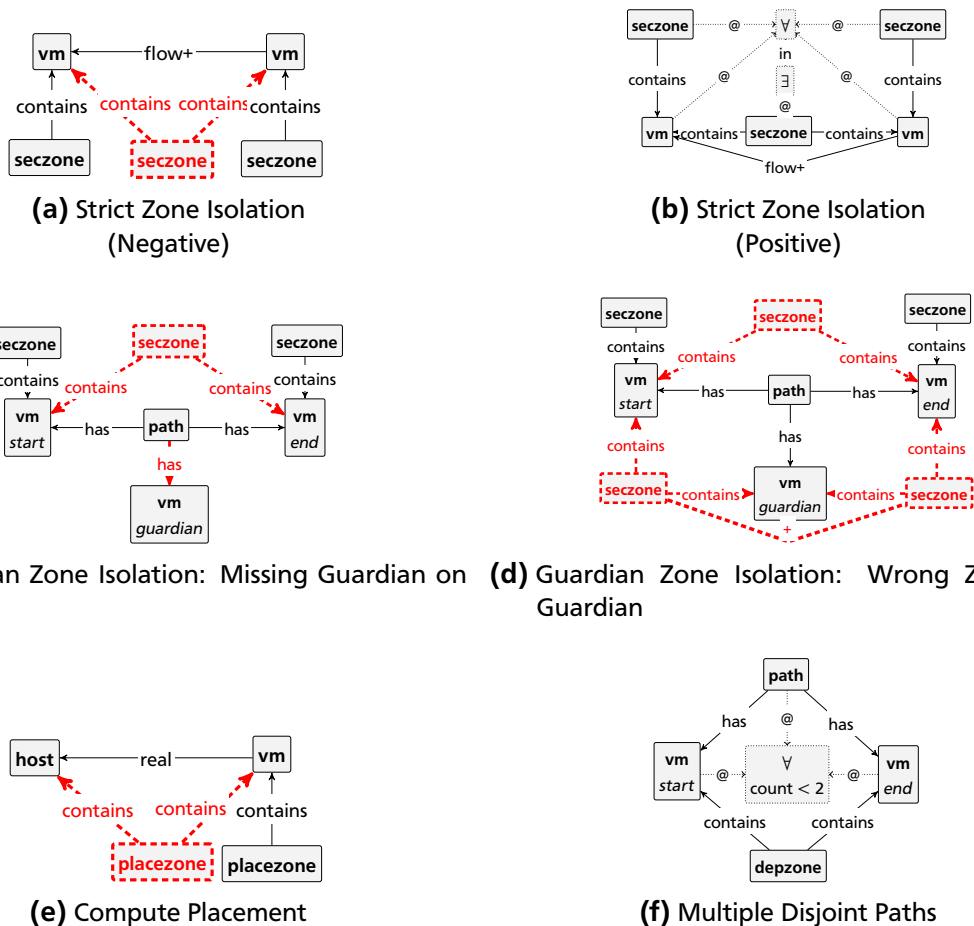
### 8.3.3 Infrastructure Policies as Graph Matches

---

The final piece of our analysis effort is the specification of security and operational policies. We formalize a wide variety of practical policies, such as isolation of security zones and prevention of single point of failures, as graph matches. Instead of production rules that transform the model, the policy rules only try to match a given graph pattern.

We usually express the security policies as *attack* states, i.e., a state of the topology or configuration that violates the desired security properties. Instead of verifying that a security property holds for the entire infrastructure, we try to find violations. However the formalism and analysis allow for both the specification of positive and negative policies. The analysis stops, i.e., finds a violation, if a propositional formula of the form  $AttackPolicy_1 \vee \neg PositivePolicy_1 \vee AttackPolicy_2 \dots$  is satisfied. That is, an attack state has been found when an attack policy has matched, or a positive policy no longer matches. Attack state policies have an advantage in the root-cause analysis of policy violations, since the analysis returns the matching part of the infrastructure that causes the violation, i.e., the attack state. Whereas for positive policies, the analysis does not provide a reason why a policy rule no longer matches.

In the following we present a subset of policies that stem from security requirements of practitioners of infrastructure cloud deployments. The policies span compute, network, and storage resources, and make use of our information flow model to determine isolation properties.



**Figure 8.4.:** A variety of Security and Operational Policies modeled in *GROOVE* as Graph Matches.

**Strict Security Zone Isolation:** We represent tenants as *security zones* which group together infrastructure elements, such as virtual machines, into zones. Each security zone is represented as a single vertex of type *seczone* with directed *contains* edges, that represent zone membership, to Realization model vertices. The zoning of elements is a policy setup performed by a security operator.

In this policy we require a *strict* isolation, i.e., no information flows, between any pair of zoned infrastructure elements that are not members of at least one common security zone. With the example of VMs as security zone members, we show both a positive and negative specification of this policy. Although we use VMs as an example, any infrastructure element can be grouped into security zones. Fig. 8.4a shows a negative/attack specification of the policy: We have a policy violation for a pair of zoned VMs that are connected by an information flow path (*flow+*) if they are *not* members of the same security zone. The statement *flow+* is a regular expression on edges and requires at least one flow edge. On the other hand, a positive specification of this policy (Fig. 8.4b) states that for all zoned VM pairs, which can communicate, there must exist at least one zone that contains both VMs.

We allow infrastructure elements to be part of multiple security zones, and our policy expects at least one common zone for element pairs with information flow. A problem arises when a multi-zoned element facilitates information flow between single-zoned elements. For example, if a VM acts as a firewall and is part of two security zones, then VMs of one zone may communicate with VMs of the other zone via the firewall VM, which is a violation of strict isolation. We deal with such inter-zone trusted elements with the *guardian* isolation policy (Section 8.3.3.1).

---

**Compute Placement:** The policy mandates the assignment of computing resources, that is on which physical hosts virtual machines must run. The motivation stems from both performance and availability reasons as well as security and legal requirements. Imagine that VMs must run on hosts from a particular geo-location due to privacy laws and data security requirements. By grouping physical hosts and VMs together into *placement zones*, similar to the previous security zones, this policy is violated if a VM is hosted on a physical server of another placement zone or no zone at all (cf. Fig. 8.4e). A zoned host can run VMs that are not part of any zone.

The related security concern of side-channel attacks due to VM co-location on the same physical host [RTSS09] is covered by the security zone isolation policy. The trust assumption if a particular hypervisor provides strong VM isolation or not is captured in the user-configurable information flow rules (cf. Section 8.3.2). From practical security policies we learned that co-locating different tenants is allowed only for a particular set of hypervisor products that are considered trusted.

**Shared Storage Isolation:** An isolation breach happens when two virtual machines of different tenants share the same storage device. We can approach this policy from two directions: Either rely on the expressive information flow analysis and the security zone isolation policy to detect such a violation, or directly find such suspicious patterns in the Realization model. We implemented the latter way which can be efficiently matched without any path finding.

---

### 8.3.3.1 Policies with Information Flow Path Conditions

---

We are also dealing with policies that have requirements on the information flow paths. For example, the guardian isolation policy requires that a trusted component, such as a firewall, is part of an information flow path between elements of different zones. With regard to single point of failures, a policy specifies the requirement of at least two fully disjoint information flow paths between two elements.

To express such policies, we can no longer rely on the *flow+* path construct, because we cannot inspect the found paths. We model an explicit path finding with traversal rules that add vertices to a *path* vertex with directed edges to denote path membership. The state exploration applies the traversal rules, which perform a graph traversal on flow edges, and constructs all possible paths between pairs of *start* and *end* nodes. We can now express policies that verify conditions on the found paths.

**Guardian Security Zone Isolation:** Given a pair of elements that are not members of a common security zone and that are connected by an information flow path. It is mandatory that the communication is mandated by a trusted guardian, i.e., a vertex flagged as *guardian* must be part of the information flow path between the pair (cf. Fig. 8.4c). Additionally, the guardian must share a security zone with each element of the pair (cf. Fig. 8.4d). The first policy is violated if there exists a path between a pair of VMs, which do not belong to a same security zone, and the path does not contain a VM flagged as *guardian*. The second policy catches the violation that a guardian VM exists on the path, but the guardian does not share a security zone with either the start or end VM. The negative edge labeled with *+* represents an OR condition for the two negative conditions of the VM and guardian zone matching.

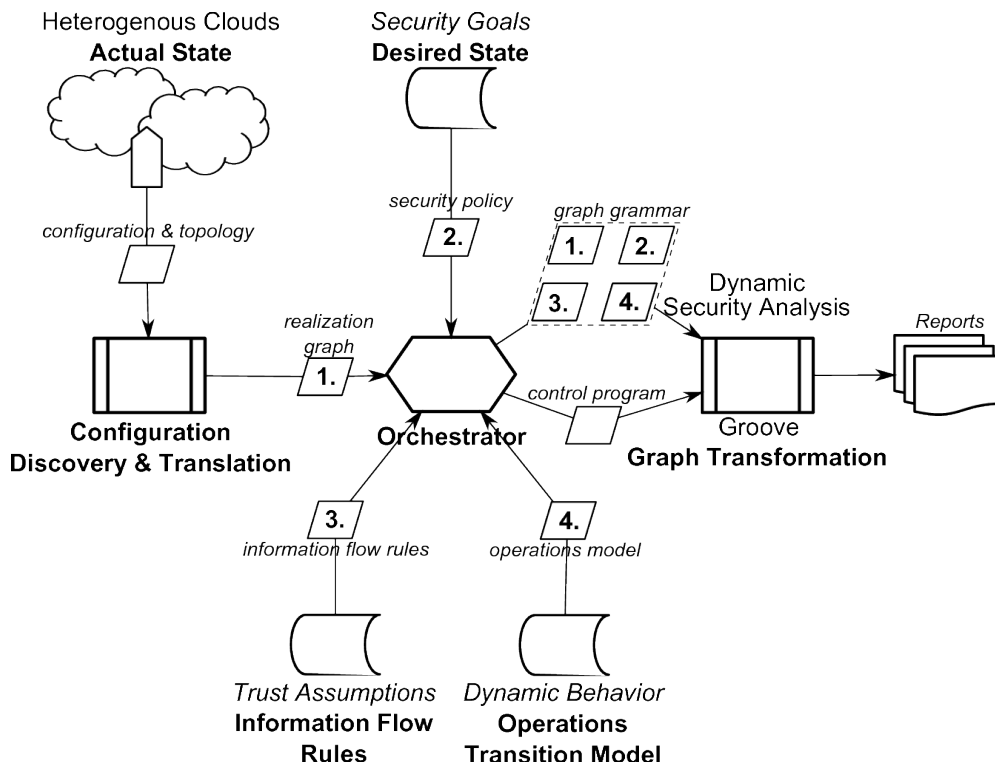
**Multiple Disjoint Paths:** We define a *dependability zone* as a group of infrastructure elements that require mutually redundant fully disjoint paths. The motivation is to prevent single point of failures between dependent infrastructure elements. Fig. 8.4f shows the corresponding rule as an attack state matching. We are using a universal quantifier with the ability of counting the number of paths between a pair of nodes of the same dependability zone (*depzone*). The policy is violated if the paths count is less than two, or any redundancy factor that is required.

### 8.3.3.2 Infrastructure Policies Summary

The formalization of policies using graph transformation and in particular also the usage of the *GROOVE* tool shows that this approach is both expressive as well as usable. We demonstrated a variety of policies ranging from zone isolation, placement of virtual machines, to the prevention of single point of failures. This covers the policy areas of isolation, operational correctness, and failure resilience that have been introduced for virtualized infrastructure policies in Chapter 4. We showed the formalization of those policies as graph matches in *GROOVE*, and further explored different ways to express policies, such as negative and positive matching. Besides an expressive and general-purpose approach, the usability is equally important so that end-users, such as security administrators or auditors of cloud environments, can specify new policies. *GROOVE* offers a graphical editor to develop new production rules, in fact the policies shown in Fig. 8.4 have been develop graphically and exported as-is. This provides an intuitive and efficient way of specifying new policies.

## 8.4 Automated Analysis: Design and Implementation

*Weatherman* provides an automated analysis of configuration and topology changes in virtualized infrastructures. Its architecture, as shown in Fig. 8.5, obtains all the necessary inputs for the analysis and invokes *GROOVE* as the graph transformation engine. Based on this architecture, we describe two application scenarios for change management (Section 8.4.3) as well as for run-time enforcement of security policies and the mitigation of misconfigurations (Section 8.4.2).



**Figure 8.5.:** The system architecture consists of i) *Configuration Discovery & Translation* on the left, which extracts the infrastructure configuration and builds the Realization model; ii) the *Orchestrator* in the middle, which prepares the graph grammar for the analysis based on all inputs; iii) and the *Graph Transformation* on the right that employs *GROOVE*.

---

## 8.4.1 System Architecture

---

The system architecture, as shown in Fig. 8.5, consists of i) *Configuration Discovery & Translation* on the left, which extracts the infrastructure configuration and builds the Realization model; ii) the *Orchestrator* in the middle, which prepares the graph grammar for the analysis based on all inputs; iii) and the *Graph Transformation* on the right that employs *GROOVE*.

The architecture is an extension to the analysis architecture originally proposed in Chapter 5. The orchestrator now also includes the user-configurable information flow rules as well as the operations transition model. We employ a different analysis backend, that is a graph transformation engine rather than set-rewriting model checkers or FOL theorem provers.

### Configuration Discovery & Translation

The configuration extraction and model population for the virtualized infrastructure is based on the method of Chapter 3 and also incorporates the updating of the model when the infrastructures changes (cf. Chapter 7). This is crucial for the run-time enforcement of security policies as the underlying infrastructure constantly changes and the change operations have to be evaluated against a model of the latest infrastructure configuration.

### Orchestrator

The orchestrator prepares the graph grammar for the analysis with *GROOVE* and includes the following elements:

- *Host Graph*: The initial graph for the graph transformation is the latest Realization model, which is serialized into an XML graph format for *GROOVE*.
- *Transformation Rules*: The transformation rules for the information flow analysis as well as for the operations transition model.
- *Graph Matches*: The security policies are expressed as graph matches, i.e., graph transformation rules that neither add nor remove elements from the graph.
- *Control Program*: The control program guides the application of the information flow and operations rules. The program specifies the order of operations as well as their parameters.

With regard to obtaining the required inputs, the involvement of the user is kept to a minimum as we are striving for an automated approach. The rules for the automated information flow analysis as well as the security policies come as pre-defined sets, and only in specific cases can/need to be extended or modified by the user. Security policies may require further input from the user, such that virtual resources need to be assigned to security zones for the zone isolation policy.

### Graph Transformation

We employ *GROOVE* for the graph transformation and analysis once the orchestrator prepared the graph grammar and control program. *GROOVE* will interpret and execute the control program for the given grammar. It will apply the information flow rules as well as the rules for the operations that are specified in the control program with their parameters.

Of particular interest is the potential violation of a given security policy once the operations have been applied and the information flow computed. Recall that a policy is violated if a propositional formula of the form  $\text{AttackPolicy}_1 \vee \neg \text{PositivePolicy}_1 \vee \text{AttackPolicy}_2 \dots$  is satisfied, i.e., either an attack policy is satisfied or a positive policy is no longer matched. *GROOVE* provides the concept of *acceptors* that indicate when such a state, a *result state*, is found. We employ different acceptors:

- *Invariant Acceptor*: Given a single transformation rule, this acceptor fires when either the rule is applicable (in positive mode) or the rule is no longer applicable (in negative mode). Note that the

---

given rule must only be applicable and not actually applied. *GROOVE* evaluates the acceptor for each new graph state. The acceptor only works for a single attack or positive security policy. The acceptor's mode has to be set to positive for attack policies and negative for positive policies, that is, the acceptor succeeds when an attack policies is applicable or when a positive rule is no longer applicable.

- *Application Acceptor*: This acceptor succeeds if a single given rule has actually been applied. This differs to the invariant acceptor where a rule only has to be applicable or not. The acceptor only works for a single attack state policy and the control program needs to explicitly try to apply the attack state rule after the operations and information flow rules have been applied.
- *Formula Acceptor*: Unlike the previous acceptors, this one can match against multiple attack and positive policies. A propositional formula, in the previously discussed form, is given and the acceptor succeeds when the formula is satisfied. Similar to the invariant acceptor, the rules must either be applicable or not.

The difference between the acceptors in terms of rule applicability and actual application of rules is important for the efficiency of our analysis, because we differentiate between a *greedy* and *non-greedy* analysis. In the greedy variant, we try to find a policy violation as early as possible and may terminate the analysis early. In the non-greedy analysis we apply the operations and information flow rules and only then determine potential policy violations. Typically, the greedy analysis is preferred for its potential early termination. However, if the current infrastructure already contains a policy violation, the intended operations are not evaluated, although they may fix the cause of the current policy violation. Therefore, we have to use a non-greedy analysis in that case.

We can instruct *GROOVE* on how many result states should be found before terminating the application of further transformation rules. For the greedy analysis with its early termination, we want to find one possible violation of a security policy. In many use cases finding a policy violation as soon as possible is enough rather than finding all possible violations. Each result state is an instance of the infrastructure where a policy is violated. As discussed previously, attack state policies have an advantage in the root-cause analysis, since the analysis returns the matching part of the infrastructure that causes the violation, i.e., the attack state. Whereas for positive policies, the analysis does not provide a reason why a policy rule no longer matches.

*GROOVE* supports multiple different graph exploration strategies, i.e., the way how the transformation rules are applied. In our case, we employ a *linear* exploration strategy, because our rule application is strictly guided by the control program. In terms of model checking, *GROOVE* also provides a full state space exploration that would analyze the interleaving of operations as well as evaluating potential operations that could have been applied and which would lead to a security policy violation. We consider this kind of analysis as future work that builds upon the modeling and analysis results of this work.

---

## 8.4.2 Run-time Analysis of Changes

---

Run-time analysis enables automated mitigation of misconfigurations and enforcement of security policies. To achieve this goal, we introduce an *authorization proxy* that acts as a reverse HTTPS proxy in front of the otherwise shielded management host. The proxy intercepts management operations and inspects them for the analysis. The proxy keeps sessions for each logged in administrators and associates the operations with them. Operations and configuration changes are only forwarded by the proxy to the management host if the *Weatherman* analysis indicates no security policy violation. In a secure deployment (cf. Section 8.5.1), it allows to protect virtualized infrastructures from malicious adversaries.

The communication in front of the manager is usually standardized: VMware and Amazon EC2 management operations are SOAP-based, whereas OpenStack is REST-based. These formats are easily inspected. In addition, the proxy tracks session states derived from the the user-login and the infrastructure manager



---

session cookie, to distinguish sessions of multiple administrators interacting with the infrastructure manager concurrently.

The Policy Decision Point (PDP) of the authorization proxy translates intercepted management operations into our operations transition model and into a change plan in the *GROOVE* control language. We have translation modules for all covered operations. For instance, from an *UpdatePortGroup* operation the proxy extracts the host, identifying port group name, new VLAN identifier, as well as new port group name. The PDP then delegates the change plan analysis to *Weatherman*, which analyzes the intended changes. The Policy Enforcement Point (PEP) only accepts the intercepted operations if they are compliant with the policies; otherwise, they are rejected. The authorization proxy refrains from forwarding the management operation in the reject case, i.e., they are not deployed in the actual infrastructure. It signals an error back to the administrator client, including the policy violation as data for diagnosis.

The run-time analysis might block management operations in two cases: Hard blocking occurs if the authorization proxy rejects a management operation, but the administrator might need to override the security policy in an emergency, which could be allowed by a trusted super-administrator. Soft blocking occurs due to the delay the analysis adds and may be precarious if the expected time between management operations is smaller than the expected analysis time.

---

### 8.4.3 Change Plan Analysis

---

The goal of the change plan analysis is to support the planning of complex configuration changes and to verify their security compliance. The focus of this complementary approach lies on the planning of potential changes and perform what-if analyses, whereas the run-time analysis inspects the operations that are currently deployed. In fact, change management, and change plans in particular, are often employed as part of IT infrastructure operation workflows and processes. In our case, an administrator drafts a sequence of desired changes that he wants to be provisioned.

The crucial question is: Will the proposed changes render the infrastructure insecure? To answer this question, the administrator submits a change plan that contains a sequence of operations with their parameters to *Weatherman*. The analysis system then uses *GROOVE* to apply the changes to the graph model of the infrastructure and verifies the resulting infrastructure state against the desired security policies. By that, the tool can establish a what-if analysis and determine what security impact the intended changes will have on the infrastructure.

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

---

## 8.5 Evaluation

---

---

### 8.5.1 Security Analysis

---

The analysis is based on the system model of Section 8.2 and the run-time analysis (Section 8.4.2): *Weatherman* is deployed with an authorization proxy (PEP) that intercepts management operations, forwards them to the policy decision point (PDP) for analysis, and which in turn issues an accept/deny decision. We establish a secure deployment that allows to obtain the integrity property based on a small set of assumptions.

- *Limited Access* [access]: The adversary accesses the virtualized infrastructure through the management interface only, which can be enforced by placing hosts into *lockdown mode* [VMw13b], where direct hypervisor configuration access is forbidden, with no privileges to revoke it. Further, this

---

implies that the adversary does neither have physical or root access on the physical hosts, direct access to the hypervisor nor physical access to network and storage. The adversary does not have access as `super_admin`, who manages the privileges. *Weatherman* and the authorization proxy are deployed in a hardened configuration and thereby placed under [access].

- *Network Isolation* [netisolation]: The management network is isolated from adversarial access, which implies that the management host cannot be accessed by the adversary directly, but only through the authorization proxy. We call the network between authorization proxy and management host  $net_{sec}$ , either enforced 1) as dedicated physical network, 2) as VLAN in the physical switch, where virtualization administrators do not have access, or 3) as a virtual network with a dedicated VLAN identifier, where the administrators do not have privileges to change it. *Weatherman* and the authorization proxy are deployed in  $net_{sec}$  and their communication with the management host is covered by [netisolation]. To strengthen this assumption further, the entities of  $net_{sec}$  communicate over secure and mutually-authenticated channels.
- *Authentic View and Faithful Model* [authenticview]: *Weatherman* has an authentic view of the topology and configuration of the infrastructure as well as a faithful model of it, including the consequences of management operations. For the infrastructure model we rely on the discovery and translation methodology of Chapter 3, in particular as discussed with regard to the security aspects in Section 3.4. For the operations we rely on the modeling approach introduced in Section 8.3.1. The Realization model provides a faithful graph representation derived from the actual configuration as the structure is encoded there. The operations model captures how individual management operations change the state of the infrastructure and thereby the Realization model.

**Definition 42** (Integrity of Run-time Analysis). *If a set of management operations  $S$  has been provisioned to the virtualized infrastructure, then Weatherman has previously verified  $S$  with respect to the specified security goals and issued an accept decision and the management host consequently provisioned  $S$ .*

We pursue the argument by back-tracking starting from a set of management operations  $S$  received at the management host.

- *Integrity of communication*: We know that the management network  $net_{sec}$  between management host, authorization proxy and analysis is covered by [netisolation] and gain integrity on  $S$  and on topology data. As the management host received  $S$  at the management network, it must have been forwarded by the authorization proxy upon an accept decision from the analysis (PDP). The analysis thereby must have verified  $S$  under the given security policy and issued an accept decision.
- *View equivalence on the topology*: From the assumption [authenticview], we obtain both the faithful Realization model of the topology and representation of consequences in the operations model as necessary conditions. Given that authentic view and faithful model, the tool can only have issued an accept decision, if none of the alarm states defined in the security policy matched the what-if state of the topology amended with the management operations of  $S$ .
- *View equivalence on  $S$* : *Weatherman* and the authorization proxy are protected from the adversary's direct influence by [access]. The management operations  $S$  are transferred between authorization proxy and *Weatherman* with integrity, by which *Weatherman* analyzes the very same  $S$  as staged for provisioning at the authorization proxy. We have that the  $S$  received at the management host must have been the same submitted at the authorization proxy and analyzed by *Weatherman*, which could only have been forwarded if a what-if analysis did not match an alarm state.
- *Exclusive provisioning through the management host*: Finally, given the [access] condition, we have that management operations can only be provisioned through the management host and that the adversary cannot access hypervisors and physical hosts directly. Thereby,  $S$  must have been provisioned by the management host itself after the verification and an accept decision.

---

## Discussion

The run-time analysis (Section 8.4.2) offers protection against malicious insiders, while the change-plan analysis (Section 8.4.3) offers non-malicious administrators a way to verify changes before provisioning. This approach benefits system availability, since honest administrators can evaluate their change plans pro-actively to gain confidence that their changes will not be denied at run-time.

The security analysis hinges on the authentic view of *Weatherman* when it comes to the topology structure and the consequences of management operations. Even though Section 8.3.1 seeks to establish a faithful representation and a systematic approach to validate the model against reality, it is still the case that “the map is not the territory”. The Realization and operations models are likely to suffer from subtle differences to the real configuration. Our systematic operations modeling and infrastructure discovery methodologies aim to reduce these differences. We further empirically evaluate the operations model in a real infrastructure environment and test the detection of expected violating operations as part of our security testing, which cover the entire workflow of change discovery, infrastructure model update, operations model, information flow analysis, and policy verification.

The effectiveness of *Weatherman*’s analysis largely depends on the quality of the input specifications: First, the information flow rules represent the trust assumptions on isolation properties and determine which components are assumed to pass on information. We employ the information flow analysis of Chapter 6, but in a simpler form without the SCC optimization. Second, *Weatherman* only finds attack states for configuration changes as provided in the specified security policy. Its analysis offers a model checking for these attack states; it does not constitute a security proof. We rely on *GROOVE* to correctly apply our graph transformation rules on the given host graph with the control program that guides the application of rules. *GROOVE* has been applied to a variety of different case studies and matured over years of development. One potential attack vector is the complexity of SOAP that can be exploited to break the proper inspection of the web service requests within the authorization proxy. Somorovsky et al. [SHJ<sup>+</sup>11] show successful attacks against the authentication of SOAP-based cloud management interfaces. However, this potential attack vector is rooted in a software vulnerability of parsing SOAP messages, not a potential inherent security flaw in our architecture. Our threat model denotes that software attacks are out of scope.

## Security Testing

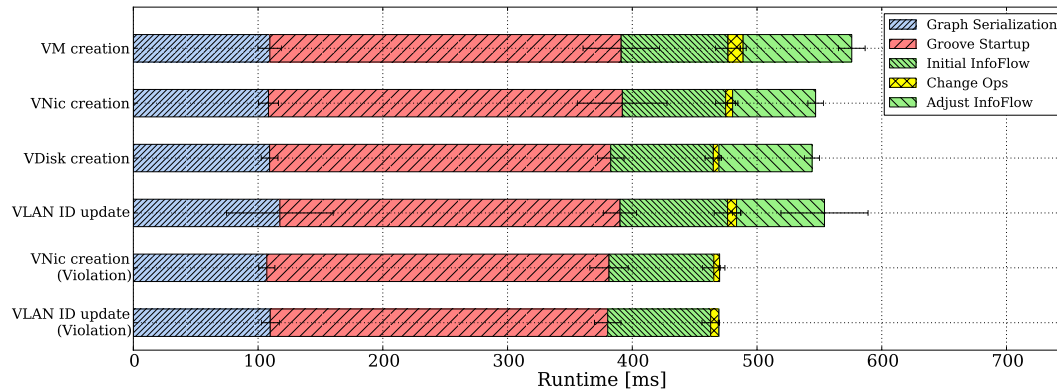
Besides arguing about the security of our system, we systematically test its ability to detect known violating operations and differentiate them from non-violating ones, on the operations set of Section 8.5.2 (cf. Fig. 8.6). For each operation, we probabilistically select parameters either from a set of violating or non-violating ones. We issue the operation to the authorization proxy with an expectation that for the violation case we obtain a *reject* decision with a particular policy violation as the reason. Otherwise, for non-violating parameters, the operation should be accepted. *Weatherman* detected all violation cases and behaves as expected. Clearly, security testing and modeling are going hand-in-hand as an iterative process, in which we make the experience that corner cases discovered in security testing serve well to improve the model and to close the maps-territory gap.

---

## 8.5.2 Performance Measurements

We empirically evaluate and discuss the performance of *Weatherman* in the case-study of a semi-production environment. We operate in the run-time analysis mode, which intercepts and analyzes operations, and we are interested in the performance of our analysis, in particular the application of the operations model and the information flow analysis.

The environment consists of 2 physical hosts and over 100 virtual machines. *Weatherman* itself runs in a Linux VM with 12 vCPUs, 12 GB RAM, and Java 1.7. We issue a variety of operations to the authorization proxy of *Weatherman*. These include the creation of virtual machines, virtual network interfaces, as well as virtual disks. Further, we change the VLAN identifier of a port group. This set of operations aligns with our subset of VMware operations (cf. Table 8.1) and covers all types of infrastructure resources as well as



**Figure 8.6.:** Time measurements for the analysis of a variety of operations, including two violating ones (the last two). We measure the times for the graph serialization, *GROOVE* start-up, initial and adjusting information flow analysis, as well as applying the change operation.

different kinds of operations. Further, we issue operations in a policy violating form to show how the analysis may stop early once a violation is found.

Fig. 8.6 shows the results of our performance evaluation with different operations on the y-axis, and the mean run-time measurement in milliseconds for 30 rounds on the x-axis. A first observation is that the majority of the analysis time for this environment is spent on serializing the graph model and initializing *GROOVE*, which loads the grammar including the graph model. This is an implementation limitation as *GROOVE* was not designed to be integrated into another application and requires to load a grammar from the filesystem. As part of the actual analysis, we observe that the times for the initial and adjusting information flow analyses are the dominant factors. Applying the operation to the graph model is almost negligible. Note that in the case of violating operations, the analysis can terminate early, i.e., not complete the adjusting information flow analysis, once a violation was found. Overall, the performance results for this environment, i.e., obtaining analysis results in under a second, are suitable for both run-time analysis and change planning.

### 8.5.2.1 Discussion on Scalability and Optimizations

We studied the scalability of *Weatherman* with a VMware infrastructure simulator, which is part of the official VMware vCenter server appliance. For a simulated environment with 1000 VMs, which resulted in a Realization model graph with 4121 vertices and 6140 edges, we obtained an overall analysis time of 253s for finding a violation in a *UpdatePortGroup* operation. This makes our approach suitable for the change plan analysis, but causes a long soft blocking for a run-time analysis. In a simulated environment with 10000 VMs (41201 vertices, 61400 edges) *GROOVE* ran out of available memory (12GB). In another case study reported by Smid and Rensink [SR13], *GROOVE* showed similar performance where in a case with 10000 elements *GROOVE* ran out of memory on a 10GB machine. A distributed variant of *GROOVE* with state compression has been proposed by Kant [Kan10], where an up to 52 times memory reduction has been achieved in one case.

In terms of performance comparison with the approach of Chapter 7, we observe for the real environments (size 100 respectively 150 VMs) a similar analysis time of 480ms for *Weatherman* respectively 476ms for *Cloud Radar* in the *initial* mode in the case of finding a VLAN ID update violation. However, in terms of scalability, *Cloud Radar* shows significant better performance for larger environments in particular in the *event-based* mode.

We stress that establishing the models, methodology, and analysis system has been the primary focus of this work, and not providing an optimized and scalable analysis. We now outline multiple directions of optimizations and scalability improvements. A short-term optimization is to reduce the size of the Realization model graph by removing nodes of types that are not addressed by production rules of the

---

grammar. Possible long-term optimizations are to transform *GROOVE* graph grammars into native code (an approach employed by GrGen [GBG<sup>+</sup>06]), to exploit a parallel processing of production rules (in particular for rules with universal quantifier and the confluent simple information flow rules), and to leverage existing large-scale graph processing framework, which however are not yet aimed for graph transformations.

---

## 8.6 Summary

---

In this chapter, we tackle the problem of misconfigurations, insider attacks and resulting security failures in virtualized infrastructures. Our solution consists of a practical tool called *Weatherman* that employs a formal model of cloud management operations, an information flow analysis to determine isolation properties, and a policy verifier in order to proactively assess infrastructure changes with regard to their security impact. For instance, we are able to detect and mitigate changes that i) break the network isolation of tenants, ii) create virtual machines in the wrong location, and iii) introduce single point of failures. We offer the run-time enforcement of security policies as well as change planning for what-if analyses. While for concreteness we focus in this work on a particular practical system and goals, we believe that our work is a first step towards a general verification methodology for virtualized infrastructures. One key aspect of our approach is the use of graph rewriting, which offers an expressive and intuitive method for formalizing the operations, information flow analysis, as well as policies.



---

## 9 Conclusions

This chapter summarizes and discusses the research contributions of this dissertation. We further highlight the key design decisions that we made, how they shaped the scope of this work, and we discuss potential directions of future work that build up on our contributions and expand on the design scope.

---

### 9.1 Research Summary

---

In this dissertation we addressed the problem that complex, scalable, and dynamic virtualized infrastructures suffer from misconfigurations, malicious insiders, and system vulnerabilities, which ultimately lead to security and operational failures, such as isolation breaches. We therefore proposed the following hypothesis in Section 1.2:

*It is possible to model and analyze complex, scalable, and dynamic virtualized infrastructures with regard to user-defined security and operational policies in an automated way.*

We developed a practical analysis framework for virtualized infrastructures that allows operators to express security and operational policies as well as to analyze and maintain the infrastructure with regard to these policies in an automated way.

---

#### 9.1.1 Overview and Comparison of Tools and Approaches

---

In Table 9.1 we provide an overview of the different tools and approaches as part of the framework. The initial tool *SAVE* analyzes a *static* snapshot of a virtualized infrastructure and adopts the concept of “colors” [Rus82] to perform a graph traversal with coloring. The policy is limited to information flow policies and we annotate information sinks with allowed colors. A “color spill” is a violation of such a policy when an information sink is colored with a non-allowed color. In Chapter 5 we study and build a tool that leverages existing model checkers and theorem provers to analyze both *static* infrastructures as well as *dynamic* aspects such as VM migration. The information flow analysis is based on graph reachability and formalized using both state transitions as well as Horn clauses. We extend the scope of the policies by using our *VALID* policy language (cf. Chapter 4). Building up on the study of dynamic aspects in virtualized infrastructures, we differentiate between a *reactive* approach that analyzes changes to the infrastructure after they have happened and a *proactive* approach that analyzes changes before they are applied. The tool *Cloud Radar* follows a reactive approach where a model of a dynamic virtualized infrastructure is maintained. Unlike the previous approaches of information flow analysis that are based on graph traversal and coloring, this tool builds an information flow graph. In particular, a *differential* flow graph that is partially re-computed for changes in the infrastructure model. Policies are expressed both in native code (Scala) as well as in a graph matching language (*GROOVE*). Finally, *Weatherman* follows a proactive approach where operations and their changes are analyzed before they are applied to the infrastructure. Building up on the policies expressed in *GROOVE* as part of *Cloud Radar*, we use *GROOVE* to express our operations transition model, the information flow analysis that builds up a information flow graph, and extend the number of policies.

#### Summary of Approach Validation and Tool Evaluation

We performed the following validation of our approaches and evaluation of our tools. For *SAVE* (Chapter 3) we analyzed the approach (Section 3.4) with regard to potential faults in the discovery and translation

**Table 9.1.:** Overview of Approaches and Tools

Tool	Approach	Information Flow	Policy
SAVE (Chapter 3)	Static	Graph traversal with colors	Color Spill
Verif (Chapter 5)	Static & Dynamic	Reachability analysis	<i>VALID</i>
<i>Cloud Radar</i> (Chapter 7)	Dynamic Reactive	Diff. Information Flow Graph	Scala & <i>GROOVE</i>
<i>Weatherman</i> (Chapter 8)	Dynamic Proactive	Information Flow Graph	<i>GROOVE</i>

phases, their mitigation in our approach, and their impact on the detection rates. Further, we reduce the correctness of the information flow approach to the correctness of the individual rules, and discuss a practical set of rules as part of a case study for a virtualized infrastructure of a financial institution (Section 3.6). The scope of our policy language *VALID* with regard to virtualized infrastructure security goals is validated as part of a case study (Section 4.2.4). We demonstrate the language by expressing a variety of policy goals (Section 4.5). For the validation of the automated verification (Chapter 5), we conducted several small case studies for different kinds of policies (zone isolation, secure migration, single point of failure). Based on a larger infrastructure example from a previous case study we evaluated the performance. We analyze the algorithms of the dynamic information flow approach (Chapter 6) with regard to termination and complexity, and compare the complexity with the approach of Chapter 3. Based on a fault model adopted from the analysis of firewall rules, we analyze the correctness of our rule ordering and application. As part of the evaluation of *Cloud Radar* (Chapter 7), we conducted performance measurements for the individual phases of the tool and for differently sized virtualized infrastructures, and compare the new approach to the full (non-differential) approach. Our security analysis covers secure deployment of the system and obtaining change events securely. Further, we perform security testing with known violating and non-violating operations to validate the detection of policy breaches. Finally, for *Weatherman* (Chapter 8) we follow a similar evaluation as for *Cloud Radar*. We measure the performance for the analysis of different operations, and we discuss the scalability of the system. Further, the security analysis also covers the deployment of the system and the integrity of the analysis. The security testing assessed the ability of the system to detect and block known violating operations, while allowing non-violating operations.

---

## 9.1.2 Summary of Research Questions

---

In the following we discuss our research questions from Section 1.2 and how these have been addressed in this dissertation.

**Q1** *How to model virtualized infrastructures with their configuration and topology? How to populate such a model in an automated way? What is the scope of the model?*

We model a virtualized infrastructure as a graph model that contains infrastructure elements, such as VMs, hypervisors, storage, network, as graph nodes with attributes that capture their configuration, and graph edges that represent the topology of the infrastructure. We populate such a model in an automated way by extracting the configuration of different virtualization systems, such as Xen, VMware, KVM, and PowerVM, and translating the different configuration formats into our unified graph model. In terms of scope, we focus our modeling on the topology and configuration of the virtualized infrastructure, and we treat VMs as “black boxes”. We do not discover nor model the configuration or state of the operating system or applications that are running inside a VM.



---

**Q2** *What is a suitable isolation and information flow model? How to determine isolation among tenants in the infrastructure?*

We propose a static information flow analysis based on the graph model representation of a virtualized infrastructure and build an automated analysis framework. The approach takes explicitly specified trust assumptions as well as security zoning of infrastructure elements, and performs a graph traversal to determine potential information flows between elements of different zones. We propose the notion of *structural information control* for a static infrastructure topology with respect to a set of trust and information flow assumptions when there does not exist inter-zone information flow unless mediated by a trusted guardian. The objective of our analysis is to reduce the complexity for a human administrator to the specification a few of those trust assumptions and let the tool extrapolate those to the entire infrastructure topology. From that and the zoning information, the tool diagnoses isolation breaches and provides refinement for a root causes analysis.

**Q3** *How to express operational and security requirements? What requirements need to be expressed? What kind of formal foundations are suitable that enable an automated analysis?*

We propose a formal security assurance language for virtualized infrastructure topologies. For our language, we study the areas deployment correctness, failure resilience, and isolation, and propose exemplary definitions for security requirements in these areas. We consider in particular operations requirements, for instance, provisioning and de-provisioning of machines or establishing dependencies, as well as security requirements, such as sufficient redundancy or isolation of tenants. We embed our assurance language in the tool-independent Intermediate Format (IF), which is well suited for automated reasoning. The language's formal foundations lie in a set-rewriting approach, commonly used in automated analysis of security protocols, with an extension to graph analysis functions.

**Q4** *How to verify that the infrastructure — given as a model — fulfills the security requirements? What are existing analysis tools? How suitable, expressive, and efficient are they?*

We built an analysis system that applies general-purpose model-checking to verify if a virtualized infrastructure satisfies security requirements given in our formal policy language. In our approach we consider both static and dynamic analysis cases, where in the static case the infrastructure is fixed and matched against given policies, and in the dynamic case where an potential attacker could modify the infrastructure. This allows us to analyze a virtualized infrastructure with regard to a variety of complex security requirements. We employ a versatile portfolio of existing problem solvers, and evaluate different analysis strategies based on Horn clauses and transition rules. We are able to analyze the infrastructure of a financial institution in a case-study using our approach with optimizations.

**Q5** *How to cope with the infrastructure's dynamic behavior? How can we keep the infrastructure model up to date? Can we efficiently analyze changes happening in the infrastructure with regard to their security impact?*

We design and implement an automated security monitoring and analysis system for dynamic virtualized infrastructures. In our approach we monitor the infrastructure for changes and update a graph model representation of the infrastructure by translating those changes into graph deltas. Furthermore, with regard to isolation security goals, we establish a static information flow analysis for dynamic system models based on dynamic information flow graphs. Compared to analysis systems that operate on static snapshots of virtualized infrastructures, our change-based approach yields significant performance improvements.

---

**Q6** *Is it possible to prevent misconfigurations in the first place? How can we model configuration and topology changes in a virtualized infrastructure? How can we analyze them?*

We model the effect of management operations on the infrastructure using graph transformations. Each operation is represented as a transformation rule that takes a graph, i.e., our graph model of the virtualized infrastructure, as an input and produces a modified graph as an output. The output graph can then be analyzed with regard to security policies that are expressed as graph matches. We built an automated system that intercepts management operations from administrators before they reach the central management host. The intercepted operations are translated into our operations model and then applied to the current state of our graph model. In case the resulting graph does violate any security policies, the operations are rejected. Otherwise, the operations are safe and forwarded to the management host, where they will actually be deployed.

---

## 9.2 Discussion and Limitations

---

The scope of our approach is defined in Section 1.3. In summary, the key scope decisions are the following.

- We aim for a static analysis allows an analysis of virtualized infrastructures without requiring infrastructure modifications. The benefits of a static analysis are that we can iterate over all possible information flows and that we can perform what-if analyses to detect policy violations. However, the analysis does not detect actual information flows in the infrastructure. A dynamic analysis would detect the actual flows, but is also limited to the monitored flows and may miss other possible flows.
- Our framework is intended to be used as a verification, monitor, and prevention tool. It is not intended to be used as a provisioning tool where a policy is used to provision a secure state nor as a remediation tool that can resolve security violations after detection. However, our prevention mechanism mitigates security violations that require remediation.
- Our analysis and modeling effort focuses on the low-level virtualized infrastructure topology including compute, network, and storage resources. In particular we focus on layer2 networking (including VLANs). We are treating routers and firewalls as abstract guardians that mediates traffic between two different security zones. We simplify the physical network topology as we observe that the configuration and complexity of the physical network is moving in the virtual network configuration.
- We treat VMs as “black boxes”, i.e., we do not perform any introspection or modeling of processes within a VM. This allows us to perform a static analysis with all the discussed advantages in our case, since we do not rely on live data through introspection.
- Our policies match against one state of the virtualized infrastructure topology and configuration. This allows infrastructure providers to ensure the compliance of their current deployment as well as the state of a future deployment through our operations model. This decision was based on typical policies of providers and the initial limitation in our discovery to perform periodic full configuration extractions.
- We assume that the software running on the configuration endpoints, for instance, the hypervisor and central management, is correct and in particular provides us with a correct and complete configuration view.

Furthermore, as a consequence of our decision to use existing general-purpose tools from the formal methods community, such as OFMC, *GROOVE*, and SPASS, we observed limitations in terms of performance and scalability of these tools. We already performed optimizations, such as graph simplification to reduce

---

the initial state size or the usage of universal quantifiers to reduce the state space. We also outlined further optimizations, for instance, the use of information flow nodes to reduce the number of information flow edges produced by complex rules.

---

### 9.3 Future Work

---

Expanding the scope of our approach and building upon the results of this dissertation leads to the following possible directions for future work.

**Integration with Access Control:** The integration of access control into our modeling and analysis is a natural extension. Existing work by Koch et al. [KMPP02] already demonstrates the successful modeling of RBAC as graphs and using graph transformations to manipulate a RBAC configuration. We can build up and extend our operations model with access control information and annotate the operations with their required privileges. Based on a set of users and their permissions, we can analyze using this extended operations model if there exists a series of operations issued by a set of users that may result into a security violation. The challenge remains to limit the search space by model checkers for the analysis. In particular, due to the possible value domains for the arguments of operations.

**Integration with Firewall Analyses:** We treat firewalls and routers as trusted guardians that mediate the information flows between two different security zones. Integration with the vast body of existing work in the area of firewall analysis would provide us stronger insights into the trustworthiness of guardians. Alternatively, we could explore the modeling of Layer3+ networking as part of our information flow analysis.

**Configuration Endpoint Attestation and Active Probes:** We assume a correct configuration endpoint without compromised or vulnerable software, which provides us with a complete and correct configuration view. As future work, we can leverage existing technologies for remote attestation, such as Intel TXT or SGX, to provide us a guarantee of the running software, i.e., to exclude that a malicious hypervisor has been loaded. Furthermore, we can employ *active* probes that not rely on the central management nodes, but which gather configuration data on their own and which are remotely attested too.

**VM Introspection and Modeling:** We are currently treating VMs as “black boxes” and do not extend our modeling into the VMs. However, this could be a useful extension, for instance, to automatically determine the zoning of a VM depending on the information and data stored or processed within a VM. We could integrate our configuration extraction and modeling with CloudFlow [BFB<sup>+</sup>14] which uses VM introspection to extract SELinux labels from within the VM. However, this would shift our approach towards a dynamic analysis with VM introspection and possibly taint tracking.

**Temporal Security Policies:** Our policies govern the current state of a virtualized infrastructure including the quantification on paths, e.g., for redundancy purposes. However, we do not cover policies that span over multiple infrastructure states or temporal security policies. Our monitoring system that records infrastructure changes over time enable the verification of temporal security goals. Existing work on temporal logic to formalize security goals has been proposed [BKM10], where Chinese wall and separation of duty policies among others have been formalized. Future work includes to study the formalization of virtualization specific policies in temporal logic and their analysis based on our monitoring system.

**Root Cause Analysis and Remediation:** Determining the root cause of a policy violation and offering remediation steps are open problems that are not fully covered by our analysis framework. The prevention mechanism in our framework will mitigate such problems. However, even if we just monitor the operations without policy enforcement, they can be used to perform correlation of actions and alarms to enable root cause analysis. We can build upon existing work in related areas, for

---

instance, Pinpoint [CKF<sup>+</sup>02] performs root-cause analysis in large and dynamic IT services through request tracing and Julisch [Jul03] proposes a clustering algorithm on alarms for root cause analysis in intrusion detection systems.

**Policy-based Provisioning:** Given that our framework already captures the current state of a virtualized infrastructure and the associated security policies, we can build a policy-based provisioning framework on top. We can leverage the root cause analysis and remediation as discussed previously to transform the current (violating) state of the infrastructure into a compliant one. Configuration synthesis through model finding has already been demonstrated for network [Nar05b] and virtualization [KH14] configurations.

**Persistent Information Flow:** Analog to extending our policies to span multiple infrastructure states, the information flow analysis has to be extended to cover *persistent* information flows by tracking infrastructure changes. For example, if a virtual disk is detached in one state and then attached to another VM in another state. Our current information flow analysis would not detect this as an isolation breach. As part of future work the information flow analysis has to incorporate taint tracking that is persistent among infrastructure states. Taint tracking has already been used on mobile devices [EGH<sup>+</sup>14] as well as in cloud environments (cf. Section 2.2). However, these are dynamic analyses that taint the live data. As we are striving for a static analysis, the taint tracking would happen in the infrastructure model, thereby not requiring any hypervisor or other infrastructure modifications.

**Lazy Information Flow:** Instead of computing and maintaining the entire information flow model, we could study a *lazy* approach where information flow is computed on demand. For example, only when a complex rule evaluation has a connectivity condition or a policy is evaluated that operates on the connectivity. This is similar to the *lazy intruder* and lazy data types used in the OFMC model checker [BMV05a], which copes with infinite state spaces.

---

## 9.4 Conclusions and Commercial Impact

---

This dissertation demonstrated that it is in fact possible to model and analyze virtualized infrastructures with regard to user-defined policies in an automated way. We build an automated security analysis framework and evaluate it in case studies with the production infrastructure of a financial institution, with a semi-production environment in a laboratory environment, as well as with simulated infrastructures. The research in this dissertation led to the creation of a new IBM product called *IBM PowerSC Trusted Surveyor* [BCD<sup>+</sup>13], which provides inventory and analysis of network isolation in IBM PowerVM-based virtualized infrastructures. Our contributions to the product line were honored by an IBM Research Division award.

---

## Bibliography

- [ABB<sup>+</sup>05] Alessandro Armando, David A. Basin, Yohan Boichut, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, Paul Hankes Drielsma, Pierre-Cyrille Héam, Olga Kouchnarenko, Jacopo Mantovani, Sebastian Mödersheim, David von Oheimb, Michaël Rusinowitch, Judson Santiago, Mathieu Turuani, Luca Viganò, and Laurent Vigneron. The avispa tool for the automated validation of internet security protocols and applications. In *CAV*, pages 281–285, 2005.
- [AC04] Alessandro Armando and Luca Compagna. SATMC: A SAT-Based Model Checker for Security Protocols. In *Logics in Artificial Intelligence*, 2004.
- [AC08] Alessandro Armando and Luca Compagna. Sat-based model-checking for security protocols analysis. *International Journal of Information Security*, 7:3–32, 2008.
- [Acc12] Accenture. Much More from Much Less: Data Center Virtualization. Available at <http://www.accenture.com/us-en/Pages/success-data-center-virtualization-summary.aspx>, 2012.
- [Aci07] Onur Aciıçmez. Yet another microarchitectural attack: exploiting i-cache. In *CSAW '07: Proceedings of the 2007 ACM workshop on Computer security architecture*, pages 11–18, New York, NY, USA, 2007. ACM.
- [AFBB02] David G. Andersen, Nick Feamster, Steve Bauer, and Hari Balakrishnan. Topology inference from bgp routing dynamics. In *Proceedings of the 2Nd ACM SIGCOMM Workshop on Internet Measurment*, IMW '02, pages 243–248, New York, NY, USA, 2002. ACM.
- [AHAS13] S. Al-Haj and E. Al-Shaer. A Formal Approach for Virtual Machine Migration Planning. In *Network and Service Management (CNSM), 2013 9th International Conference on*, pages 51–58, Oct 2013.
- [AJ11] A. Arefin and Guofei Jiang. CloudInsight: Shedding Light on the Cloud. In *Reliable Distributed Systems (SRDS), 2011 30th IEEE Symposium on*, pages 219–228, Oct 2011.
- [AKM<sup>+</sup>14] Yossi Azar, Seny Kamara, Ishai Menache, Mariana Raykova, and Bruce Shepard. Co-location-resistant clouds. In *Proceedings of the 6th Edition of the ACM Workshop on Cloud Computing Security, CCSW '14*, pages 9–20, New York, NY, USA, 2014. ACM.
- [AKS07] Onur Aciıçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. In *ASIACCS '07: Proceedings of the 2nd ACM symposium on Information, computer and communications security*, pages 312–320, New York, NY, USA, 2007. ACM.
- [ALRL04] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11 – 33, jan.-march 2004.
- [Ama] Amazon Web Services. Trusted advisor. <https://aws.amazon.com/premiumsupport/trustedadvisor/>.
- [Ama11] Amazon Web Services. Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region. <http://aws.amazon.com/message/65648/>, April 2011.

- 
- [Ama14a] Amazon Web Services. AWS Config. Available at <https://aws.amazon.com/config/>, 2014.
- [Ama14b] Amazon Web Services. The Amazon Elastic Compute Cloud (EC2). Available at <https://aws.amazon.com/ec2/>, 2014.
- [ANSZ09] Ahmed M. Azab, Peng Ning, Emre C. Sezer, and Xiaolan Zhang. HIMA: A Hypervisor-Based Integrity Measurement Agent. In *Proceedings of the 2009 Annual Computer Security Applications Conference, ACSAC '09*, pages 461–470, Washington, DC, USA, 2009. IEEE Computer Society.
- [ANW<sup>+</sup>10] Ahmed M. Azab, Peng Ning, Zhi Wang, Xuxian Jiang, Xiaolan Zhang, and Nathan C. Skalsky. HyperSentry: Enabling Stealthy In-context Measurement of Hypervisor Integrity. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS '10*, pages 38–49, New York, NY, USA, 2010. ACM.
- [AS86] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. Technical report, Cornell University, Ithaca, NY, USA, 1986.
- [ASMEAE08] Ehab Al-Shaer, Will Marrero, Adel El-Atawy, and Khalid ElBadawi. Global Verification and Analysis of Network Access Control Configuration. Technical report, DePaul University, 2008.
- [ASMEAE09] E. Al-Shaer, W. Marrero, A. El-Atawy, and K. Elbadawi. Network configuration in a box: towards end-to-end verification of network reachability and security. In *Network Protocols, 2009. ICNP 2009. 17th IEEE International Conference on*, pages 123–132, Oct 2009.
- [AU95] Alfred V. Aho and Jeffrey D. Ullman. *Foundations of Computer Science*. Computer Science Press, W. H. Freeman and Company, 1995.
- [AVA07] AVANTSSAR. Requirements for modeling and ASLan v.1. Deliverable D2.1, Automated Validation of Trust and Security of Service-oriented Architectures (AVANTSSAR), 2007. <http://www.avantssar.eu/pdf/deliverables/avantssar-d2-1.pdf>.
- [AVA10] AVANTSSAR. ASLan final version with dynamic service and policy composition. Deliverable D2.3, Automated Validation of Trust and Security of Service-oriented Architectures (AVANTSSAR), 2010. <http://www.avantssar.eu/pdf/deliverables/avantssar-d2-3.pdf>.
- [AVI03] AVISPA. The Intermediate Format. Deliverable D2.3, Automated Validation of Internet Security Protocols and Applications (AVISPA), 2003. <http://www.avispa-project.org/delivs/2.3/d2-3.pdf>.
- [AWK02] Paul Ammann, Duminda Wijesekera, and Saket Kaushik. Scalable, Graph-based Network Vulnerability Analysis. In *Proceedings of the 9th ACM Conference on Computer and Communications Security, CCS '02*, pages 217–224, New York, NY, USA, 2002. ACM.
- [AWY08] Richard Alimi, Ye Wang, and Y. Richard Yang. Shadow configuration as a network management primitive. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*. ACM, 2008.
- [Axe00] Stefan Axelsson. The base-rate fallacy and the difficulty of intrusion detection. *ACM Trans. Inf. Syst. Secur.*, 3(3):186–205, 2000.
- [BBCL11] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, and Carlos Luna. Formally verifying isolation and availability in an idealized model of virtualization. In *Proceedings of the 17th International Conference on Formal Methods, FM'11*, pages 231–245, Berlin, Heidelberg, 2011. Springer-Verlag.

- 
- [BBGR03] Y. Bejerano, Y. Breitbart, Minos Garofalakis, and Rajeev Rastogi. Physical topology discovery for large multisubnet networks. In *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies*, volume 1, pages 342–352 vol.1, March 2003.
- [BBI<sup>+</sup>13] Sören Bleikertz, Sven Bugiel, Hugo Ideler, Stefan Nürnberger, and Ahmad-Reza Sadeghi. Client-controlled Cryptography-as-a-Service in the Cloud. In *International Conference on Applied Cryptography and Network Security (ACNS 2013)*. Springer, Jun 2013.
- [BBKL13] T. Binz, U. Breitenbucher, O. Kopp, and F. Leymann. Automated discovery and maintenance of enterprise topology graphs. In *Service-Oriented Computing and Applications (SOCA), 2013 IEEE 6th International Conference on*, pages 126–134, Dec 2013.
- [BCD<sup>+</sup>13] Axel Buecker, Fernando Costa, Rosa Davidson, Enrico Matteotti, Geraint North, David Sherwood, and Simon Zaccak. *Managing Security and Compliance in Cloud or Virtualized Data Centers Using IBM PowerSC*, chapter Trusted Surveyor. IBM Redbooks, 2013.
- [BCG<sup>+</sup>06] Stefan Berger, Ramón Cáceres, Kenneth A. Goldman, Ronald Perez, Reiner Sailer, and Leendert van Doorn. vTPM: Virtualizing the Trusted Platform Module. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS’06, Berkeley, CA, USA, 2006. USENIX Association.
- [BCP<sup>+</sup>08] Stefan Berger, Ramón Cáceres, Dimitrios Pendarakis, Reiner Sailer, Enriquillo Valdez, Ronald Perez, Wayne Schildhauer, and Deepa Srinivasan. Tvdc: managing security in the trusted virtual datacenter. *SIGOPS Oper. Syst. Rev.*, 42:40–47, January 2008.
- [BCQ<sup>+</sup>13] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. DepSky: Dependable and Secure Storage in a Cloud-of-Clouds. *Trans. Storage*, 9(4):12:1–12:33, November 2013.
- [BDD<sup>+</sup>11] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Stephan Heuser, Ahmad-Reza Sadeghi, and Bhargava Shastry. Practical and Lightweight Domain Isolation on Android. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, SPSM ’11*, pages 51–62, New York, NY, USA, 2011. ACM.
- [BDD<sup>+</sup>12] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. Towards Taming Privilege-Escalation Attacks on Android. In *19th Annual Network & Distributed System Security Symposium (NDSS’12)*, Feb 2012.
- [BDF<sup>+</sup>03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, SOSP ’03*, pages 164–177, New York, NY, USA, 2003. ACM.
- [Bel05] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of USENIX Annual Technical Conference*, pages 41–46, 2005.
- [Bel08] Fabrice Bellard. Qemu accelerator technical documentation. Internet, May 2008.
- [BEP<sup>+</sup>14] J. Bacon, D. Eyers, T.F.J.-M. Pasquier, J. Singh, I. Papagiannis, and P. Pietzuch. Information flow control for secure cloud computing. *Network and Service Management, IEEE Transactions on*, 11(1):76–89, March 2014.
- [BF07] Hitesh Ballani and Paul Francis. Conman: a step towards network manageability. In *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols*

- 
- for computer communications, SIGCOMM '07, pages 205–216, New York, NY, USA, 2007. ACM.
- [BFB<sup>+</sup>14] Mirza Basim Baig, Connor Fitzsimons, Suryanarayanan Balasubramanian, Radu Sion, and Donald E. Porter. Cloudflow: Cloud-wide policy enforcement using fast vm introspection. In *Proceedings of the 2014 IEEE International Conference on Cloud Engineering, IC2E '14*, pages 159–164, Washington, DC, USA, 2014. IEEE Computer Society.
- [BFG10] Moritz Y. Becker, Cédric Fournet, and Andrew D. Gordon. Secpal: Design and semantics of a decentralized authorization language. *J. Comput. Secur.*, 18(4):619–665, December 2010.
- [BFL<sup>+</sup>12] Tobias Binz, Christoph Fehling, Frank Leymann, Alexander Nowak, and David Schumm. Formalizing the cloud through enterprise topology graphs. In *Proceedings of the 2012 IEEE Fifth International Conference on Cloud Computing, CLOUD '12*, pages 742–749, Washington, DC, USA, 2012. IEEE Computer Society.
- [BG11] Sören Bleikertz and Thomas Groß. A Virtualization Assurance Language for Isolation and Deployment. In *IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY'11)*. IEEE, Jun 2011.
- [BGJ<sup>+</sup>04] Yuri Breitbart, Minos Garofalakis, Ben Jai, Cliff Martin, Rajeev Rastogi, and Avi Silberschatz. Topology discovery in heterogeneous ip networks: The netinventory system. *IEEE/ACM Trans. Netw.*, 12(3):401–414, June 2004.
- [BGM11] Sören Bleikertz, Thomas Groß, and Sebastian Mödersheim. Automated Verification of Virtualized Infrastructures. In *ACM Cloud Computing Security Workshop (CCSW'11)*. ACM, Oct 2011.
- [BGM16] Sören Bleikertz, Thomas Groß, and Sebastian Mödersheim. Dynamic Information Flow Graphs with Flow Rules. Technical Report RZ3893, IBM Research, 2016.
- [BGMV15] Sören Bleikertz, Thomas Groß, Sebastian Mödersheim, and Carsten Vogel. Proactive Security Analysis of Changes in Virtualized Infrastructures. In *Annual Computer Security Applications Conference (ACSAC 2015)*. ACM, Dec 2015.
- [BGSE11] Sören Bleikertz, Thomas Groß, Matthias Schunter, and Konrad Eriksson. Automated Information Flow Analysis of Virtualized Infrastructures. In *16th European Symposium on Research in Computer Security (ESORICS'11)*. Springer, Sep 2011.
- [BGV14] Sören Bleikertz, Thomas Groß, and Carsten Vogel. Cloud Radar: Near Real-Time Detection of Security Failures in Dynamic Virtualized Infrastructures. In *Annual Computer Security Applications Conference (ACSAC 2014)*. ACM, Dec 2014.
- [BHJ09] Mathieu Bastian, Sebastien Heymann, and Mathieu Jacomy. Gephi: An open source software for exploring and manipulating networks. In *International AAAI Conference on Weblogs and Social Media*, 2009.
- [BHK12] Abdeltouab Belbekkouche, Md. Mahmud Hasan, and Ahmed Karmouch. Resource discovery and allocation in network virtualization. *Communications Surveys Tutorials, IEEE*, 14(4):1114–1128, Fourth 2012.
- [Bib77] K. J. Biba. Integrity considerations for secure computer systems. Technical report, MITRE Corp., 04 1977.



- 
- [BK85] W. E. Boebert and R. Y. Kain. A practical alternative to hierarchical integrity policies. In *Proceedings of the 8th National Computer Security Conference*, 1985.
- [BKF<sup>+</sup>11] J. Bellessa, E. Kroske, R. Farivar, M. Montanari, K. Larson, and R.H. Campbell. NetODESSA: Dynamic Policy Enforcement in Cloud Networks. In *Reliable Distributed Systems Workshops (SRDSW), 2011 30th IEEE Symposium on*, pages 57–61, Oct 2011.
- [BKM10] David Basin, Felix Klaedtke, and Samuel Müller. Monitoring Security Policies with Metric First-order Temporal Logic. In *Proceedings of the 15th ACM Symposium on Access Control Models and Technologies, SACMAT '10*, pages 23–34, New York, NY, USA, 2010. ACM.
- [BKNS12] Sören Bleikertz, Anil Kurmus, Zoltan A. Nagy, and Matthias Schunter. Secure Cloud Maintenance - Protecting workloads against insider attacks. In *7th ACM Symposium on Information, Computer and Communications Security (ASIACCS'12)*. ACM, May 2012.
- [BKS14] KhalidZaman Bijon, Ram Krishnan, and Ravi Sandhu. A formal model for isolation management in cloud infrastructure-as-a-service. In ManHo Au, Barbara Carminati, and C.-C.Jay Kuo, editors, *Network and System Security*, volume 8792 of *Lecture Notes in Computer Science*, pages 41–53. Springer International Publishing, 2014.
- [Bla01] Bruno Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society.
- [BLCSG12] Shakeel Butt, H. Andrés Lagar-Cavilla, Abhinav Srivastava, and Vinod Ganapathy. Self-service cloud computing. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security, CCS '12*, pages 253–264, New York, NY, USA, 2012. ACM.
- [BLP76] E. D. Bell and J. L. La Padula. Secure computer system: Unified exposition and multics interpretation, 1976.
- [BMNW04] Yair Bartal, Alain Mayer, Kobbi Nissim, and Avishai Wool. Firmato: A novel firewall management toolkit. *ACM Trans. Comput. Syst.*, 22(4):381–420, November 2004.
- [BMP<sup>+</sup>13] Sören Bleikertz, Toni Mastelić, Sebastian Pape, Wolter Pieters, and Trajce Dimkov. Defining the Cloud Battlefield - Supporting Security Assessments by Cloud Customers. In *IEEE International Conference on Cloud Engineering (IC2E 2013)*. IEEE, Mar 2013.
- [BMV05a] David Basin, Sebastian Mödersheim, and Luca Viganò. OFMC: A symbolic model checker for security protocols. *International Journal of Information Security*, 4(3):181–208, June 2005. Published online December 2004.
- [BMV05b] David A. Basin, Sebastian Mödersheim, and Luca Viganò. OFMC: A symbolic model checker for security protocols. *Int. J. Inf. Sec.*, 4(3):181–208, 2005.
- [BN89] D.F.C. Brewer and M.J. Nash. The chinese wall security policy. In *Security and Privacy, 1989. Proceedings., 1989 IEEE Symposium on*, pages 206–214, May 1989.
- [BSP<sup>+</sup>10] Sören Bleikertz, Matthias Schunter, Christian W. Probst, Konrad Eriksson, and Dimitrios Pendarakis. Security Audits of Multi-tier Virtual Infrastructures in Public Infrastructure Clouds. In *ACM Cloud Computing Security Workshop (CCSW'10)*. ACM, Oct 2010.
- [Bur95] Mark Burgess. A Site Configuration Engine. *Computing Systems*, 8(2):309–337, 1995.
- [CCL<sup>+</sup>13] Mel Cordero, Lucio Correia, Hai Lin, Vamshikrishna Thatikonda, Rodrigo Xavier, and Scott Vetter. *IBM PowerVM Virtualization Introduction and Configuration*. IBM Redbooks, 2013.

- 
- [CDE<sup>+</sup>10] Luigi Catuogno, Alexandra Dmitrienko, Konrad Eriksson, Dirk Kuhlmann, Gianluca Ramunno, Ahmad-Reza Sadeghi, Steffen Schulz, Matthias Schunter, Marcel Winandy, and Jing Zhan. Trusted virtual domains: Design, implementation and lessons learned. In *Proceedings of the First International Conference on Trusted Systems, INTRUST'09*, pages 156–179, Berlin, Heidelberg, 2010. Springer-Verlag.
- [CDRS07] Serdar Cabuk, Chris I. Dalton, HariGovind Ramasamy, and Matthias Schunter. Towards automated provisioning of secure virtualized networks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security, CCS '07*, pages 235–245, New York, NY, USA, 2007. ACM.
- [CDZ97] K.L. Calvert, M.B. Doar, and E.W. Zegura. Modeling internet topology. *Communications Magazine, IEEE*, 35(6):160–163, June 1997.
- [cfe] Cfengine. <http://www.cfengine.com>.
- [CGMH14] David Cock, Qian Ge, Toby Murray, and Gernot Heiser. The Last Mile: An Empirical Study of Timing Channels on seL4. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, CCS '14*, pages 570–581, New York, NY, USA, 2014. ACM.
- [che] Chef. <http://www.opscode.com/chef/>.
- [Cit13] Citrix. XenServer Management API, Jun 2013. [http://docs.vmd.citrix.com/XenServer/6.2.0/1.0/en\\_gb/api/](http://docs.vmd.citrix.com/XenServer/6.2.0/1.0/en_gb/api/).
- [CKF<sup>+</sup>02] M.Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 595–604, 2002.
- [CKG<sup>+</sup>08] A. Caracas, A. Kind, D. Gantenbein, S. Fussenegger, and D. Dechouniotis. Mining semantic relations using netflow. In *Business-driven IT Management, 2008. BDIM 2008. 3rd IEEE/IFIP International Workshop on*, pages 110–111, April 2008.
- [CLHX10] Fei Chen, Alex X. Liu, JeeHyun Hwang, and Tao Xie. First Step Towards Automatic Correction of Firewall Policy Faults. In *Proceedings of the 24th International Conference on Large Installation System Administration, LISA'10*, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.
- [CLHX12] Fei Chen, Alex X. Liu, Jeehyun Hwang, and Tao Xie. First Step Towards Automatic Correction of Firewall Policy Faults. *ACM Trans. Auton. Adapt. Syst.*, 7(2):27:1–27:24, July 2012.
- [CMVdM09] Xu Chen, Z. Morley Mao, and Jacobus Van der Merwe. PACMAN: A Platform for Automated and Controlled Network Operations and Configuration Management. In *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies*. ACM, 2009.
- [CN12] William R. Claycomb and Alex Nicoll. Insider Threats to Cloud Computing: Directions for New Research Challenges. In *Proceedings of the 2012 IEEE 36th Annual Computer Software and Applications Conference, COMPSAC '12*, pages 387–394, Washington, DC, USA, 2012. IEEE Computer Society.
- [CPK10] Yanpei Chen, Vern Paxson, and Randy H. Katz. What's new about cloud computing security? Technical Report UCB/EECS-2010-5, EECS Department, University of California, Berkeley, Jan 2010.

- 
- [CSA10] CSA. Top Threats to Cloud Computing V1.0. Technical report, Cloud Security Alliance (CSA), mar 2010.
- [CSA13] CSA. The Notorious Nine: Cloud Computing Threats in 2013. Technical report, Cloud Security Alliance (CSA), Feb 2013.
- [CSRL01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.
- [CW87] D. D. Clark and D. R. Wilson. A comparison of commercial and military computer security policies. In *IEEE Symposium on Security and Privacy*, pages 184–195. IEEE Computer Society, 1987.
- [CZMB08] Xu Chen, Ming Zhang, Z. Morley Mao, and Paramvir Bahl. Automating network application dependency discovery: Experiences, limitations, and new solutions. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI'08*, pages 117–130, Berkeley, CA, USA, 2008. USENIX Association.
- [DBSL02] Nicodemos C. Damianou, Arosha K. Bandara, Morris S. Sloman, and Emil C. Lupu. A survey of policy specification approaches. Technical report, Imperial College of Science Technology and Medicine, 2002.
- [DDLS01] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The ponder policy specification language. In *Proceedings of the International Workshop on Policies for Distributed Systems and Networks, POLICY '01*, pages 18–38, London, UK, 2001. Springer-Verlag.
- [Den76] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, May 1976.
- [DeT02] John DeTreville. Binder, a logic-based security language. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy, SP '02*, pages 105–, Washington, DC, USA, 2002. IEEE Computer Society.
- [DF07] B. Donnet and T. Friedman. Internet topology discovery: a survey. *Communications Surveys Tutorials, IEEE*, 9(4):56–69, Fourth 2007.
- [DGCQ13] Adrian Duncan, Michael Goldsmith, Sadie Creese, and James Quinton. Cloud Computing: Insider Attacks on Virtual Machines During Migration. In *The 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (IEEE TrustCom'13)*. IEEE, 2013.
- [DJ07] Thomas Delaet and Wouter Joosen. PoDIM: A Language for High-Level Configuration Management. In *Proceedings of the 21st conference on Large Installation System Administration Conference, LISA'07*, pages 21:1–21:13, Berkeley, CA, USA, 2007. USENIX Association.
- [DMT10] DMTF. Open virtualization format specification. Technical report, DMTF, 2010.
- [EGC<sup>+</sup>10] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, OSDI'10*, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.
- [EGH<sup>+</sup>14] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *ACM Trans. Comput. Syst.*, 32(2):5:1–5:29, June 2014.

- 
- [EJDP14] A. Eghesadi, Y. Jarraya, M. Debbabi, and M. Pourzandi. Preservation of Security Configurations in the Cloud. In *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, pages 17–26, March 2014.
- [EKC98] Michael Ernst, Craig Kaplan, and Craig Chambers. Predicate Dispatching: A Unified Theory of Dispatch. In *Proceedings of the 12th European Conference on Object-Oriented Programming, ECCOP '98*, pages 186–211, London, UK, UK, 1998. Springer-Verlag.
- [EMC10] EMC IT. EMC IT's Journey to the Private Cloud: Server Virtualization. Available at <http://www.emc.com/collateral/software/white-papers/h8135-it-journey-server-virtualization-wp.pdf>, 2010.
- [EMS<sup>+</sup>07] William Enck, Patrick Drew McDaniel, Subhabrata Sen, Panagiotis Sebos, Sylke Spoerel, Albert G. Greenberg, Sanjay G. Rao, and William Aiello. Configuration Management at Massive Scale: System Design and Experience. In *USENIX Annual Technical Conference*, 2007.
- [ENI09] ENISA. Cloud computing: Benefits, risks and recommendations for information security. Technical report, European Network and Information Security Agency (ENISA), nov 2009.
- [ETS12] ETSI. Network Functions Virtualisation, Oct 2012. [https://portal.etsi.org/nfv/nfv\\_white\\_paper.pdf](https://portal.etsi.org/nfv/nfv_white_paper.pdf).
- [FKBE15] Falzon, Kevin, Bodden, and Eric. Dynamically Provisioning Isolation in Hierarchical Architectures. In Javier Lopez and Chris J. Mitchell, editors, *Information Security*, volume 9290 of *Lecture Notes in Computer Science*, pages 83–101. Springer International Publishing, September 2015. Awarded best student paper.
- [FLH<sup>+</sup>00] D. Farinacci, T. Li, S. Hanks, D. Meyer, and P. Traina. Generic Routing Encapsulation (GRE). RFC 2784, 2000.
- [FM11] Leo Freitas and John McDermott. Formal methods for security in the xenon hypervisor. *Int. J. Softw. Tools Technol. Transf.*, 13(5):463–489, October 2011.
- [FS08] Liana Fong and Malgorzata Steinder. Duality of Virtualization: Simplification and Complexity. *SIGOPS Oper. Syst. Rev.*, 42(1):96–97, January 2008.
- [GBG<sup>+</sup>06] Rubino Geiß, Gernot Veit Batz, Daniel Grund, Sebastian Hack, and Adam Szalkowski. GrGen: A Fast SPO-Based Graph Rewriting Tool. In Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg, editors, *Third International Conference on Graph Transformation (ICGT 2006)*, volume 4178 of *Lecture Notes in Computer Science*, pages 383–397. Springer, 2006.
- [GdR<sup>+</sup>11] Amir Hossein Ghamarian, Maarten de Mol, Arend Rensink, Eduardo Zambon, and Maria Zimakova. Modelling and analysis using GROOVE. *International Journal on Software Tools for Technology Transfer (STTT)*, March 2011.
- [GL12] Afshar Ganjali and David Lie. Auditing cloud management using information flow tracking. In *Proceedings of the Seventh ACM Workshop on Scalable Trusted Computing, STC '12*, pages 79–84, New York, NY, USA, 2012. ACM.
- [GM82] Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20. IEEE, 1982.

- 
- [GMTA10] T. Grandison, E.M. Maximilien, S. Thorpe, and A. Alba. Towards a Formal Definition of a Computing Cloud. In *Services (SERVICES-1), 2010 6th World Congress on*, pages 191–192, July 2010.
- [GPC<sup>+</sup>03] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A Virtual Machine-Based Platform for Trusted Computing. *SIGOPS Oper. Syst. Rev.*, 37(5):193–206, 2003.
- [GR05] Tal Garfinkel and Mendel Rosenblum. When Virtual is Harder than Real: Security Challenges in Virtual Machine Based Computing Environments. In *HOTOS'05: Proceedings of the 10th conference on Hot Topics in Operating Systems*, pages 20–20, Berkeley, CA, USA, 2005. USENIX Association.
- [Gra85] Jim Gray. Why Do Computers Stop and What Can Be Done About It? Technical Report 85.7, Tandem Computers, 1985.
- [Gra91] James W. Gray, III. Toward a mathematical foundation for information flow security. In *IEEE Symposium on Security and Privacy*, pages 21–35. IEEE, 1991.
- [Hag13] Sebastian Hagen. *Algorithms for the Efficient Verification and Planning of Information Technology Change Operations*. PhD thesis, Technische Universität München, 2013.
- [HB09] Urs Hoelzle and Luiz Andre Barroso. *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 1st edition, 2009.
- [HGC<sup>+</sup>08] Timothy L. Hinrichs, Natasha Gude, Martin Casado, John C. Mitchell, and Scott Shenker. Expressing and enforcing flow-based network security policies language. Technical report, University of Chicago, 2008.
- [HGC<sup>+</sup>09] Timothy L. Hinrichs, Natasha S. Gude, Martin Casado, John C. Mitchell, and Scott Shenker. Practical declarative network management. In *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking, WREN '09*, pages 1–10, New York, NY, USA, 2009. ACM.
- [HH02] R. Housley and S. Hollenbeck. EtherIP: Tunneling Ethernet Frames in IP Datagrams. RFC 3378, 2002.
- [Hin99] Susan Hinrichs. Policy-Based Management: Bridging the Gap. In *Proceedings of the 15th Annual Computer Security Applications Conference, ACSAC '99*, pages 209–, Washington, DC, USA, 1999. IEEE Computer Society.
- [HL12] Weili Han and Chang Lei. Survey paper: A survey on policy languages in network and security management. *Comput. Netw.*, 56(1):477–489, January 2012.
- [HLC<sup>+</sup>13] Jianan Hao, Yang Liu, Wentong Cai, Guangdong Bai, and Jun Sun. vtrust: A formal modeling and verification framework for virtualization systems. In *Formal Methods and Software Engineering*, volume 8144 of *Lecture Notes in Computer Science*, pages 329–346. Springer Berlin Heidelberg, 2013.
- [HLMS10] Fang Hao, T. V. Lakshman, Sarit Mukherjee, and Haoyu Song. Secure cloud computing with a virtualized network infrastructure. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing, HotCloud'10*, pages 16–16, Berkeley, CA, USA, 2010. USENIX Association.
- [HNB11] Brian Hay, Kara Nance, and Matt Bishop. Storm clouds rising: Security challenges for iaas cloud computing. In *Proceedings of the 2011 44th Hawaii International Conference on System Sciences, HICSS '11*, pages 1–7, Washington, DC, USA, 2011. IEEE Computer Society.

- 
- [Hol91] Gerard J. Holzmann. *Design and validation of computer protocols*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.
- [Hol97] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, May 1997.
- [HPL98] James A. Hoagland, Raju Pandey, and Karl N. Levitt. Security policy specification using a graphical approach. Technical report, University of California, Davis, 1998.
- [HPMC02] Bradley Huffak, Daniel Plummer, David Moore, and K Claffy. Topology discovery by active probing. In *Proceedings of the 2002 Symposium on Applications and the Internet (SAINT) Workshops*, SAINT-W '02, pages 90–, Washington, DC, USA, 2002. IEEE Computer Society.
- [HSK12] S. Hagen, M. Seibold, and A Kemper. Efficient verification of IT change operations or: How we could have prevented Amazon’s cloud outage. In *Network Operations and Management Symposium (NOMS), 2012 IEEE*, pages 368–376, April 2012.
- [HSUW13] Amir Herzberg, Haya Shulman, Johanna Ullrich, and Edgar Weippl. Cloudoscopy: Services discovery and topology mapping. In *Proceedings of the 2013 ACM Workshop on Cloud Computing Security Workshop, CCSW '13*, pages 113–122, New York, NY, USA, 2013. ACM.
- [HY86] J. Thomas Haigh and William D. Young. Extending the non-interference version of MLS for SAT. In *IEEE Symposium on Security and Privacy*, pages 60–60. IEEE, 1986.
- [IBM12] IBM Global Technology Services. Data center operational efficiency best practices. Available at <http://www.ibm.com/services/us/en/it-services/data-center-efficiency-study.html>, 2012.
- [Jac90] J. Jacob. Separability and the detection of hidden channels. *Inf. Process. Lett.*, 34:27–29, February 1990.
- [Jac02] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11:256–290, April 2002.
- [JED<sup>+</sup>12a] Y. Jarraya, A. Eghtesadi, M. Debbabi, Y. Zhang, and M. Pourzandi. Cloud Calculus: Security verification in elastic cloud computing platform. In *Collaboration Technologies and Systems (CTS), 2012 International Conference on*, pages 447–454, May 2012.
- [JED<sup>+</sup>12b] Yosr Jarraya, Arash Eghtesadi, Mourad Debbabi, Ying Zhang, and Makan Pourzandi. Formal Verification of Security Preservation for Migrating Virtual Machines in the Cloud. In *Proceedings of the 14th International Conference on Stabilization, Safety, and Security of Distributed Systems, SSS'12*, pages 111–125, Berlin, Heidelberg, 2012. Springer-Verlag.
- [JPRD10] Nikolai Joukov, Birgit Pfitzmann, HariGovind V. Ramasamy, and Murthy V. Devarakonda. Application-storage discovery. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference, SYSTOR '10*, pages 19:1–19:14, New York, NY, USA, 2010. ACM.
- [JSS06] Trent Jaeger, Reiner Sailer, and Umesh Shankar. PRIMA: Policy-reduced Integrity Measurement Architecture. In *Proceedings of the Eleventh ACM Symposium on Access Control Models and Technologies, SACMAT '06*, pages 19–28, New York, NY, USA, 2006. ACM.
- [Jul03] Klaus Julisch. Clustering Intrusion Detection Alarms to Support Root Cause Analysis. *ACM Trans. Inf. Syst. Secur.*, 6(4):443–471, November 2003.
- [Kan10] Gijs Kant. Distributed state space generation for graphs up to isomorphism. Master’s thesis, University of Twente, 2010.

- 
- [KBAF11] Hyojoon Kim, Theophilus Benson, Aditya Akella, and Nick Feamster. The Evolution of Network Configuration: A Tale of Two Campuses. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference, IMC '11*, pages 499–514, New York, NY, USA, 2011. ACM.
- [KCZ<sup>+</sup>13] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real Time Network Policy Checking Using Header Space Analysis. In *10th USENIX Symposium on Networked Systems Design and Implementation*, pages 99–111, Berkeley, CA, 2013. USENIX.
- [KEH<sup>+</sup>09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles, SOSP '09*, pages 207–220, New York, NY, USA, 2009. ACM.
- [KF91] N. L. Kelem and R. J. Feiertag. A Separation Model for Virtual Machine Monitors. In *IEEE Symposium on Security and Privacy*, pages 78–86. IEEE, 1991.
- [KFJ03] Lalana Kagal, Tim Finin, and Anupam Joshi. A policy language for a pervasive computing environment. In *Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks, POLICY '03*, pages 63–, Washington, DC, USA, 2003. IEEE Computer Society.
- [KGE06] A. Kind, D. Gantenbein, and H. Etoh. Relationship discovery with netflow to enable business-driven it management. In *Business-Driven IT Management, 2006. BDIM '06. The First IEEE/IFIP International Workshop on*, pages 63–70, April 2006.
- [KH14] S. Kikuchi and K. Hiraishi. Improving reliability in management of cloud computing infrastructure by formal methods. In *Network Operations and Management Symposium (NOMS), 2014 IEEE*, pages 1–7, May 2014.
- [KHK14] Safwan Mahmud Khan, Kevin W. Hamlen, and Murat Kantarcioglu. Silver lining: Enforcing secure information flow at the cloud edge. In *Proceedings of the 2014 IEEE International Conference on Cloud Engineering, IC2E '14*, pages 37–46, Washington, DC, USA, 2014. IEEE Computer Society.
- [Kik13] Shinji Kikuchi. *Improving Reliability in Management of Cloud Computing Infrastructure by Formal Methods*. PhD thesis, Japan Advanced Institute of Science and Technology, 2013.
- [KKL<sup>+</sup>07] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the Linux Virtual Machine Monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, Ottawa, Ontario, Canada, June 2007.
- [KL09] Amir R. Khakpour and Alex Liu. Quarnet: A Tool for Quantifying Static Network Reachability. Technical Report MSU-CSE-09-2, Department of Computer Science, Michigan State University, East Lansing, Michigan, January 2009.
- [KMPP02] Manuel Koch, Luigi V. Mancini, and Francesco Parisi-Presicce. A Graph-based Formalism for RBAC. *ACM Trans. Inf. Syst. Secur.*, 5(3):332–365, August 2002.
- [Kra09] F. John Krautheim. Private Virtual Infrastructure for Cloud Computing. In *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing, HotCloud'09*, Berkeley, CA, USA, 2009. USENIX Association.

- 
- [KSRL10] Eric Keller, Jakub Szefer, Jennifer Rexford, and Ruby B. Lee. Nohype: Virtualized cloud infrastructure without the virtualization. In *Proceedings of the 37th Annual International Symposium on Computer Architecture, ISCA '10*, pages 350–361, New York, NY, USA, 2010. ACM.
- [KSS<sup>+</sup>09] Sunil D. Krothapalli, Xin Sun, Yu-Wei E. Sung, Suan Aik Yeo, and Sanjay G. Rao. A toolkit for automating and visualizing vlan configuration. In *SafeConfig '09: Proceedings of the 2nd ACM workshop on Assurable and usable security configuration*, pages 63–70, New York, NY, USA, 2009. ACM.
- [Kun10] Vivek Kundra. State of Public Sector Cloud Computing, May 2010.
- [KVCP97] P. Krishna, N. H. Vaidya, M. Chatterjee, and D. K. Pradhan. A cluster-based approach for routing in dynamic networks. *SIGCOMM Comput. Commun. Rev.*, 27(2):49–64, April 1997.
- [KYB<sup>+</sup>07] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard os abstractions. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles, SOSP '07*, pages 321–334, New York, NY, USA, 2007. ACM.
- [KZZ<sup>+</sup>13] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *10th USENIX Symposium on Networked Systems Design and Implementation*, pages 15–27, Berkeley, CA, 2013. USENIX.
- [Lam73] Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.
- [Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng.*, 3:125–143, March 1977.
- [Lam90] Leslie Lamport. A theorem on atomicity in distributed algorithms. *Distributed Computing*, 4:59–68, 1990.
- [Lam02] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
- [LDL<sup>+</sup>08] Scott Loveland, Eli M. Dow, Frank LeFevre, Duane Beyer, and Phil F. Chan. Leveraging virtualization to optimize high-availability system configurations. *IBM Systems Journal*, 47(4):591–604, 2008.
- [Lip75] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.
- [LOG01] Bruce Lowekamp, David O'Hallaron, and Thomas Gross. Topology discovery for large ethernet networks. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, SIGCOMM '01*, pages 237–248, New York, NY, USA, 2001. ACM.
- [LS89] Leslie Lamport and Fred B. Schneider. Pretending atomicity. Technical Report Research Report 44, Digital Equipment Corporation, Systems Research Center, 1989.
- [Man01] Heiko Mantel. Information flow control and applications - bridging a gap. In José Nuno Oliveira and Pamela Zave, editors, *FME*, volume 2021 of *Lecture Notes in Computer Science*, pages 153–172. Springer, 2001.



- 
- [MDD<sup>+</sup>14] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright. Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks. RFC 7348, 2014.
- [MG09a] Peter Mell and Tim Grance. Effectively and Securely Using the Cloud Computing Paradigm, October 2009.
- [MG09b] Peter Mell and Tim Grance. The NIST Definition of Cloud Computing, October 2009.
- [MGHW09] Jeanna Matthews, Tal Garfinkel, Christofer Hoff, and Jeff Wheeler. Virtual machine contracts for datacenter and cloud computing environments. In *Proceedings of the 1st workshop on Automated control for datacenters and clouds*, ACDC '09, pages 25–30, New York, NY, USA, 2009. ACM.
- [Mic12] Microsoft. Windows Azure Service Interruption in Western Europe Resolved, Root Cause Analysis Coming Soon. <http://azure.microsoft.com/blog/2012/07/27/windows-azure-service-interruption-in-western-europe-resolved-root-cause-analysis-coming-soon/>, July 2012.
- [MIT14a] MIT Technology Review. Three Questions with Amazon's Technology Chief, Werner Vogels. <http://www.technologyreview.com/news/528471/three-questions-with-amazon-technology-chief-werner-vogels/>, June 2014.
- [Mit14b] Hitoshi Mitake. sheepdog: software defined storage system for converged infrastructure, May 2014. [https://sheepdog.github.io/sheepdog/\\_static/sheepdog\\_COJ14.pdf](https://sheepdog.github.io/sheepdog/_static/sheepdog_COJ14.pdf).
- [MJM<sup>+</sup>16] Suryadipta Majumdar, Yosr Jarraya, Taous Madi, Amir Alimohammadifar, Makan Pourzandi, Lingyu Wang, and Mourad Debbabi. Proactive verification of security compliance for clouds through pre-computation: Application to openstack. In Ioannis Askoxylakis, Sotiris Ioannidis, Sokratis Katsikas, and Catherine Meadows, editors, *Computer Security – ESORICS 2016: 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I*, pages 47–66, Cham, 2016. Springer International Publishing.
- [MK05] Robert Marmorstein and Phil Kearns. A Tool for Automated iptables Firewall Analysis. In *ATEC '05: Proceedings of the USENIX Annual Technical Conference*, pages 44–44, Berkeley, CA, USA, 2005. USENIX Association.
- [MKLT13] Michael Menzel, Markus Klems, Hoàng Anh Le, and Stefan Tai. A configuration crawler for virtual appliances in compute clouds. In *Proceedings of the 2013 IEEE International Conference on Cloud Engineering, IC2E '13*, pages 201–209, Washington, DC, USA, 2013. IEEE Computer Society.
- [ML97] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles, SOSP '97*, pages 129–142, New York, NY, USA, 1997. ACM.
- [MLQ<sup>+</sup>10] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. Trustvisor: Efficient tcb reduction and attestation. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy, SP '10*, pages 143–158, Washington, DC, USA, 2010. IEEE Computer Society.
- [MMB<sup>+</sup>12] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, and Gerwin Klein. Non-interference for Operating System Kernels. In *Proceedings of the Second International Conference on Certified Programs and Proofs, CPP'12*, pages 126–142, Berlin, Heidelberg, 2012. Springer-Verlag.

- 
- [MMB<sup>+</sup>13] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. seL4: From General Purpose to a Proof of Information Flow Enforcement. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 415–429, Washington, DC, USA, 2013. IEEE Computer Society.
- [MMH08] Derek Gordon Murray, Grzegorz Milos, and Steven Hand. Improving xen security through disaggregation. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '08, pages 151–160, New York, NY, USA, 2008. ACM.
- [MML<sup>+</sup>12] John McDermott, Bruce Montrose, Margery Li, James Kirby, and Myong Kang. Separation Virtual Machine Monitors. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 419–428, New York, NY, USA, 2012. ACM.
- [MMW<sup>+</sup>16] Taous Madi, Suryadipta Majumdar, Yushun Wang, Yosr Jarraya, Makan Pourzandi, and Lingyu Wang. Auditing security compliance of the virtualized infrastructure in the cloud: Application to openstack. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, CODASPY '16, pages 195–206, New York, NY, USA, 2016. ACM.
- [MRF11] Yogesh Mundada, Anirudh Ramachandran, and Nick Feamster. Silverline: Data and network isolation for cloud services. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'11, pages 13–13, Berkeley, CA, USA, 2011. USENIX Association.
- [MRF<sup>+</sup>13] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing software-defined networks. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 1–14, Berkeley, CA, USA, 2013. USENIX Association.
- [MS10] David Molnar and Stuart Schechter. Self hosting vs. cloud hosting: Accounting for the security impact of hosting in the cloud. In *Proceedings of the Ninth Workshop on the Economics of Information Security (WEIS)*, June 2010.
- [MWA02] Ratul Mahajan, David Wetherall, and Tom Anderson. Understanding BGP Misconfiguration. In *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '02, pages 3–16, New York, NY, USA, 2002. ACM.
- [MWZ00] Alain Mayer, Avishai Wool, and Elisha Ziskind. Fang: A Firewall Analysis Engine. In *SP '00: Proceedings of the 2000 IEEE Symposium on Security and Privacy*, page 177, Washington, DC, USA, 2000. IEEE.
- [Nar05a] Sanjai Narain. Network configuration management via model finding. In *Proceedings of the 19th conference on Large Installation System Administration Conference - Volume 19*, LISA '05, pages 15–15, Berkeley, CA, USA, 2005. USENIX Association.
- [Nar05b] Sanjai Narain. Network Configuration Management via Model Finding. In *Proceedings of the 19th conference on Large Installation System Administration Conference - Volume 19*, LISA '05, pages 15–15, Berkeley, CA, USA, 2005. USENIX Association.
- [NCPT06] Sanjai Narain, Y.-H. Alice Cheng, Alex Poylisher, and Rajesh Talpade. Network single point of failure analysis via model finding. In *Proceedings of First Alloy Workshop*, 2006.
- [New14] Chris Newcombe. Why Amazon Chose TLA+. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 4th International Conference, ABZ 2014, Toulouse, France, June 2-6, 2014. Proceedings*, pages 25–39, 2014.

- 
- [NMCS10] J. Nogueira, M. Melo, J. Carapinha, and S. Sargento. A distributed approach for virtual network discovery. In *GLOBECOM Workshops (GC Wkshps), 2010 IEEE*, pages 277–282, Dec 2010.
- [NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.
- [NNS02] Flemming Nielson, Hanne Riis Nielson, and Helmut Seidl. A succinct solver for alfp. *Nordic J. of Computing*, 9:335–372, December 2002.
- [NRZ<sup>+</sup>15] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon Web Services Uses Formal Methods. *Commun. ACM*, 58(4):66–73, March 2015.
- [OAS05] OASIS. eXtensible Access Control Markup Language (XACML) Version 2.0, February 2005.
- [OAS13] OASIS. Topology and Orchestration Specification for Cloud Applications Version (TOSCA) v1, Nov 2013.
- [OCJ08] Jon Oberheide, Evan Cooke, and Farnam Jahanian. Exploiting Live Virtual Machine Migration. In *BlackHat DC Briefings*, Washington DC, February 2008.
- [OGA05] Xinming Ou, Sudhakar Govindavajhala, and Andrew W. Appel. Mulval: A logic-based network security analyzer. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, pages 8–8, Berkeley, CA, USA, 2005. USENIX Association.
- [OGP03] David Oppenheimer, Archana Ganapathi, and David A. Patterson. Why do internet services fail, and what can be done about it? In *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*, USITS'03, Berkeley, CA, USA, 2003. USENIX Association.
- [Olt13] Jon Oltsik. 2013 vormetric/esg insider threats survey, Sep 2013.
- [Ope13] OpenStack. AMQP and Nova, Dec 2013. <http://docs.openstack.org/developer/nova/devref/rpc.html>.
- [Ope14a] Openstack Consortium. AgregateMultiTenancyIsolation description incorrect. <https://bugs.launchpad.net/openstack-manuals/+bug/1328400>, 2014.
- [Ope14b] Openstack Consortium. nova-manage creates network with wrong vlanid. <https://bugs.launchpad.net/nova/+bug/1288609>, 2014.
- [Orm07] Tavis Ormandy. An Empirical Study into the Security Exposure of Hosts of Hostile Virtualized Environments, 2007.
- [PdCL07] B.D. Payne, M.D.P. de Carbone, and Wenke Lee. Secure and flexible monitoring of virtual machines. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 385–397, Dec 2007.
- [Pea11] Siani Pearson. Towards Accountability in the Cloud. Technical Report HPL-2011-138, HP Laboratories, 2011.
- [Per05] Colin Percival. Cache missing for fun and profit, May 2005.

- 
- [PKZ<sup>+</sup>13] Vasilis Pappas, Vasileios P. Kemerlis, Angeliki Zavou, Michalis Polychronakis, and Angelos D. Keromytis. Cloudfence: Data flow tracking as a cloud service. In Salvatore J. Stolfo, Angelos Stavrou, and Charles V. Wright, editors, *Research in Attacks, Intrusions, and Defenses*, volume 8145 of *Lecture Notes in Computer Science*, pages 411–431. Springer Berlin Heidelberg, 2013.
- [PLS<sup>+</sup>14] Gábor Pék, Andrea Lanzi, Abhinav Srivastava, Davide Balzarotti, Aurélien Francillon, and Christoph Neumann. On the Feasibility of Software Attacks on Commodity Virtual Machine Monitors via Direct Device Assignment. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '14*, pages 305–316, New York, NY, USA, 2014. ACM.
- [PMOK<sup>+</sup>14] Christian Priebe, Divya Muthukumaran, Dan O' Keeffe, David Eyers, Brian Shand, Ruediger Kapitza, and Peter Pietzuch. Cloudsafety-net: Detecting data leakage between cloud tenants. In *Proceedings of the 6th Edition of the ACM Workshop on Cloud Computing Security, CCSW '14*, pages 117–128, New York, NY, USA, 2014. ACM.
- [PS98] Cynthia Phillips and Laura Painton Swiler. A Graph-based System for Network-vulnerability Analysis. In *Proceedings of the 1998 Workshop on New Security Paradigms, NSPW '98*, pages 71–79, New York, NY, USA, 1998. ACM.
- [pup] Puppet. <http://puppetlabs.com>.
- [PYK<sup>+</sup>10] Lucian Popa, Minlan Yu, Steven Y. Ko, Sylvia Ratnasamy, and Ion Stoica. Cloudpolice: Taking access control out of the network. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks, Hotnets-IX*, pages 7:1–7:6, New York, NY, USA, 2010. ACM.
- [PZH13] Michael Pearce, Sherali Zeadally, and Ray Hunt. Virtualization: Issues, Security Threats, and Solutions. *ACM Comput. Surv.*, 45(2):17:1–17:39, March 2013.
- [RA00] Ronald W. Ritchey and Paul Ammann. Using Model Checking to Analyze Network Vulnerabilities. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 156–, Washington, DC, USA, 2000. IEEE Computer Society.
- [Rac14] RackSpace. Managed cloud and Dedicated IaaS Leader. Available at <http://www.rackspace.com>, 2014.
- [RC11] Francisco Rocha and Miguel Correia. Lucy in the sky without diamonds: Stealing confidential data in the cloud. In *Proceedings of the 1st International Workshop on Dependability of Clouds, Data Centers and Virtual Computing Environments (DCDV, with DSN'11)*, June 2011.
- [Red10] RedHat. Application Development Guide, Aug 2010. <http://libvirt.org/guide/html/index.html>.
- [RIF02] Matei Ripeanu, Adriana Iamnitchi, and Ian Foster. Mapping the gnutella network. *IEEE Internet Computing*, 6(1):50–57, January 2002.
- [RK09] A. Rensink and J-H. Kuperus. Repotting the geraniums: on nested graph transformation rules. In A. Boronat and R. Heckel, editors, *Graph transformation and visual modelling techniques, York, U.K.*, volume 18 of *Electronic Communications of the EASST*. EASST, 2009.
- [RNSE09] Himanshu Raj, Ripal Nathuji, Abhishek Singh, and Paul England. Resource management for isolation enhanced cloud services. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security, CCSW '09*, pages 77–84, New York, NY, USA, 2009. ACM.

- 
- [Roz97] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations*, volume 1. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997.
- [RTSS09] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*, pages 199–212, New York, NY, USA, 2009. ACM.
- [Rus81] John Rushby. Design and verification of secure systems. In *Proceedings of the eighth ACM Symposium on Operating Systems Principles, SOSP '81*, pages 12–21, New York, NY, USA, 1981. ACM.
- [Rus82] John Rushby. Proof of separability a verification technique for a class of security kernels. In Mariangiola Dezani-Ciancaglini and Ugo Montanari, editors, *International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 352–367. Springer, 1982.
- [Rus92] John Rushby. Noninterference, transitivity, and channel-control security policies. Technical report, SRI International, Dec 1992.
- [RVJ09] Sandra Rueda, Hayawardh Vijayakumar, and Trent Jaeger. Analysis of Virtual Machine System Policies. In *Proceedings of the 14th ACM Symposium on Access Control Models and Technologies, SACMAT '09*, pages 227–236, New York, NY, USA, 2009. ACM.
- [RY10] Thomas Ristenpart and Scott Yilek. When Good Randomness Goes Bad: Virtual Machine Reset Vulnerabilities and Hedging Deployed Cryptography. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2010.
- [RZ04] Liam Roditty and Uri Zwick. A Fully Dynamic Reachability Algorithm for Directed Graphs with an Almost Linear Update Time. In *Proceedings of the Thirty-sixth Annual ACM Symposium on Theory of Computing, STOC '04*, pages 184–191. ACM, 2004.
- [RZFG99] Carlos Ribeiro, André Zúquete, Paulo Ferreira, and Paulo Guedes. Spl: An access control language for security policies with complex constraints. In *Proceedings of the Network and Distributed System Security Symposium*, pages 89–107, 1999.
- [SCC10] Nikhil Swamy, Juan Chen, and Ravi Chugh. Enforcing Stateful Authorization and Information Flow Policies in Fine. In *ESOP 2010: 19th European Symposium on Programming*. Springer Verlag, March 2010.
- [SCH08] Nikhil Swamy, B.J. Corcoran, and M. Hicks. Fable: A Language for Enforcing User-defined Security Policies. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 369–383, May 2008.
- [SG12] Seungwon Shin and Guofei Gu. CloudWatcher: Network security monitoring using Open-Flow in dynamic cloud networks (or: How to provide security monitoring as a service in clouds?). In *Network Protocols (ICNP), 2012 20th IEEE International Conference on*, pages 1–6, Oct 2012.
- [SGG12] Ilari Shafer, Snorri Gylfason, and Gregory R. Ganger. vQuery: a Platform for Connecting Configuration and Performance. *VMware Technical Journal*, 1(2), December 2012.
- [SGR09] Nuno Santos, Krishna P. Gummadi, and Rodrigo Rodrigues. Towards Trusted Cloud Computing. In *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing, HotCloud'09*, Berkeley, CA, USA, 2009. USENIX Association.

- 
- [SHJ<sup>+</sup>11] Juraj Somorovsky, Mario Heiderich, Meiko Jensen, Jörg Schwenk, Nils Gruschka, and Luigi Lo Iacono. All Your Clouds are Belong to us – Security Analysis of Cloud Management Interfaces. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop, CCSW '11*, New York, NY, USA, 2011. ACM.
- [SJV<sup>+</sup>05] Reiner Sailer, Trent Jaeger, Enriquillo Valdez, Ramon Caceres, Ronald Perez, Stefan Berger, John Linwood Griffin, and Leendert van Doorn. Building a mac-based security architecture for the xen open-source hypervisor. In *Proceedings of the 21st Annual Computer Security Applications Conference*, pages 276–285, Washington, DC, USA, 2005. IEEE Computer Society.
- [SK10] Udo Steinberg and Bernhard Kauer. Nova: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European Conference on Computer Systems, EuroSys '10*, pages 209–222, New York, NY, USA, 2010. ACM.
- [SM03] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21:2003, 2003.
- [SMV<sup>+</sup>10] Joshua Schiffman, Thomas Moyer, Hayawardh Vijayakumar, Trent Jaeger, and Patrick McDaniel. Seeding Clouds with Trust Anchors. In *Proceedings of the 2010 ACM Workshop on Cloud Computing Security Workshop, CCSW '10*, pages 43–46, New York, NY, USA, 2010. ACM.
- [SMWA04] Neil Spring, Ratul Mahajan, David Wetherall, and Thomas Anderson. Measuring isp topologies with rocketfuel. *IEEE/ACM Trans. Netw.*, 12(1):2–16, February 2004.
- [Sof14] SoftLayer. Cloud Servers, Storage, Big Data, & More IAAS Solutions. Available at <http://www.softlayer.com>, 2014.
- [SR13] Wietse Smid and Arend Rensink. Class diagram restructuring with GROOVE. In P. van Gorp, L.M. Rose, and C. Krause, editors, *Proceedings Sixth Transformation Tool Contest*, volume 135 of *Electronic proceedings in theoretical computer science*, pages 83–87. arXiv.org, November 2013.
- [SRGS12] Nuno Santos, Rodrigo Rodrigues, Krishna P. Gummadi, and Stefan Saroiu. Policy-sealed Data: A New Abstraction for Building Trusted Cloud Services. In *Proceedings of the 21st USENIX Conference on Security Symposium, Security'12*, pages 10–10, Berkeley, CA, USA, 2012. USENIX Association.
- [SS04] Ahmad-Reza Sadeghi and Christian Stübke. Property-based Attestation for Computing Platforms: Caring About Properties, Not Mechanisms. In *Proceedings of the 2004 Workshop on New Security Paradigms, NSPW '04*, pages 67–77, New York, NY, USA, 2004. ACM.
- [SSVJ13] Joshua Schiffman, Yuqiong Sun, Hayawardh Vijayakumar, and Trent Jaeger. Cloud Verifier: Verifiable Auditing Service for IaaS Clouds. In *Proceedings of the IEEE 1st International Workshop on Cloud Security Auditing (CSA 2013)*, June 2013.
- [SVJ12] Joshua Schiffman, Hayawardh Vijayakumar, and Trent Jaeger. Verifying System Integrity by Proxy. In *5th International Conference on Trust and Trustworthy Computing*, pages 179–201, 2012.
- [SZJvD04] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13, SSYM'04*, pages 16–16, Berkeley, CA, USA, 2004. USENIX Association.

- 
- [Tar72] Robert Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1972.
- [TPT<sup>+</sup>10] Hanghang Tong, B. Aditya Prakash, Charalampos Tsourakakis, Tina Eliassi-Rad, Christos Faloutsos, and Duen Horng Chau. On the Vulnerability of Large Graphs. In *Proceedings of the 2010 IEEE International Conference on Data Mining, ICDM '10*, pages 1091–1096, Washington, DC, USA, 2010. IEEE Computer Society.
- [Tur06] Mathieu Turuani. The cl-atse protocol analyser. In Frank Pfenning, editor, *Term Rewriting and Applications*, volume 4098 of *Lecture Notes in Computer Science*, pages 277–286. Springer Berlin / Heidelberg, 2006.
- [TZV<sup>+</sup>08] Mukarram Tariq, Amgad Zeitoun, Vytautas Valancius, Nick Feamster, and Mostafa Ammar. Answering What-if Deployment and Configuration Questions with Wise. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication, SIGCOMM '08*, pages 99–110, New York, NY, USA, 2008. ACM.
- [VKF12] Andreas Voellmy, Hyojoon Kim, and Nick Feamster. Procera: A language for high-level reactive network control. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks, HotSDN '12*, pages 43–48, New York, NY, USA, 2012. ACM.
- [VMw06] VMware. Providing LUN Security. Available at [http://www.vmware.com/pdf/esx\\_lun\\_security.pdf](http://www.vmware.com/pdf/esx_lun_security.pdf), March 2006.
- [VMw07] VMware. VMware Virtual Networking Concepts, Jul 2007. [http://www.vmware.com/files/pdf/virtual\\_networking\\_concepts.pdf](http://www.vmware.com/files/pdf/virtual_networking_concepts.pdf).
- [VMw08] VMware. The Architecture of VMware ESXi, Oct 2008. [http://www.vmware.com/files/pdf/ESXi\\_architecture.pdf](http://www.vmware.com/files/pdf/ESXi_architecture.pdf).
- [VMw09] VMware. Securing the Cloud: A Review of Cloud Computing, Security Implications and Best Practices, 2009.
- [VMw11] VMware. vSphere 5.0 API Reference, Aug 2011. [http://pubs.vmware.com/vsphere-50/topic/com.vmware.wssdk.apiref.doc\\_50/right-pane.html](http://pubs.vmware.com/vsphere-50/topic/com.vmware.wssdk.apiref.doc_50/right-pane.html).
- [VMw13a] VMware. vSphere 5.5 API Reference, Sep 2013. <http://pubs.vmware.com/vsphere-55/index.jsp#com.vmware.wssdk.apiref.doc/right-pane.html>.
- [VMw13b] VMware. vSphere Security, ESXi 5.5, vCenter Server 5.5 (EN-001164-04), 2013.
- [VMw15] VMware. VMware Virtual SAN 6.0, Feb 2015. [http://www.vmware.com/files/pdf/products/vsan/VMware\\_Virtual\\_SAN\\_Whats\\_New.pdf](http://www.vmware.com/files/pdf/products/vsan/VMware_Virtual_SAN_Whats_New.pdf).
- [VMw16] VMware. Common Criteria Evaluation & Validation (CCEVS). Available at <http://www.vmware.com/ch/security/certifications/common-criteria>, 2016.
- [Vog] Sebastian Vogl. Secure hypervisors.
- [Vuk10] Marko Vukolić. The Byzantine Empire in the Intercloud. *SIGACT News*, 41(3):105–111, September 2010.
- [WAHS10] Ruoyu Wu, Gail-Joon Ahn, Hongxin Hu, and M. Singhal. Information flow control in cloud computing. In *Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom), 2010 6th International Conference on*, pages 1–7, Oct 2010.

- 
- [WBM<sup>+</sup>06] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A Scalable, High-performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.
- [WDF<sup>+</sup>09] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. Spass version 3.5. In Renate Schmidt, editor, *Automated Deduction CADE-22*, volume 5663 of *Lecture Notes in Computer Science*, pages 140–145. Springer Berlin / Heidelberg, 2009.
- [WGR<sup>+</sup>09] Timothy Wood, Alexandre Gerber, K. K. Ramakrishnan, Prashant Shenoy, and Jacobus Van der Merwe. The case for enterprise-ready virtual private clouds. In *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*, HotCloud'09, Berkeley, CA, USA, 2009. USENIX Association.
- [WJ10] Zhi Wang and Xuxian Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 380–395, Washington, DC, USA, 2010. IEEE Computer Society.
- [WL06] Zhenghong Wang and Ruby B. Lee. Covert and Side Channels Due to Processor Architecture. In *Proceedings of the 22Nd Annual Computer Security Applications Conference*, ACSAC '06, pages 473–482, Washington, DC, USA, 2006. IEEE Computer Society.
- [WLB07] Sage A. Weil, Andrew W. Leung, Scott A. Brandt, and Carlos Maltzahn. RADOS: A Scalable, Reliable Storage Service for Petabyte-scale Storage Clusters. In *Proceedings of the 2Nd International Workshop on Petascale Data Storage: Held in Conjunction with Supercomputing '07*, PDSW '07, pages 35–44, New York, NY, USA, 2007. ACM.
- [Woj08] Rafal Wojtczuk. Adventures with a certain Xen vulnerability (in the PVFB backend). <http://invisiblethingslab.com/pub/xenfb-adventures-10.pdf>, October 2008.
- [Woo01] Avishai Wool. Architecting the Lumeta Firewall Analyzer. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, pages 7–7, Berkeley, CA, USA, 2001. USENIX Association.
- [WWGJ12] Zhi Wang, Chiachih Wu, Michael Grace, and Xuxian Jiang. Isolating commodity hosted hypervisors with hyperlock. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 127–140, New York, NY, USA, 2012. ACM.
- [WZA<sup>+</sup>09] Jinpeng Wei, Xiaolan Zhang, Glenn Ammons, Vasanth Bala, and Peng Ning. Managing Security of Virtual Machine Images in a Cloud Environment. In *Proceedings of the ACM Workshop on Cloud Computing Security*, CCSW '09, pages 91–96. ACM, 2009.
- [XZM<sup>+</sup>04] Geoffrey G. Xie, Jibin Zhan, David A. Maltz, Hui Zhang, Albert Greenberg, Gisli Hjalmytsson, and Jennifer Rexford. On Static Reachability Analysis of IP Networks, 2004.
- [YTEM06] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. *ACM Trans. Comput. Syst.*, 24:393–423, November 2006.
- [ZBWK06] Nikolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in histar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 263–278, Berkeley, CA, USA, 2006. USENIX Association.



- 
- [ZCCZ11] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. Cloudvisor: Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, SOSP '11*, pages 203–216, New York, NY, USA, 2011. ACM.
- [Zim88] H. Zimmermann. In C. Partridge, editor, *Innovations in Internetworking*, chapter OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection, pages 2–9. Artech House, Inc., Norwood, MA, USA, 1988.
- [ZJOR11] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K. Reiter. Homealone: Co-residency detection in the cloud via side-channel analysis. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy, SP '11*, pages 313–328, Washington, DC, USA, 2011. IEEE Computer Society.
- [ZJRR12] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In *Proceedings of the 2012 ACM conference on Computer and communications security, CCS '12*, pages 305–316. ACM, 2012.
- [ZJRR14] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in paas clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, CCS '14*, pages 990–1003, New York, NY, USA, 2014. ACM.
- [ZR13] Yinqian Zhang and Michael K. Reiter. Düppel: Retrofitting Commodity Operating Systems to Mitigate Cache Side Channels in the Cloud. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, CCS '13*, pages 827–838, New York, NY, USA, 2013. ACM.



---

# A Wissenschaftlicher Werdegang

## **Juli 2011 - Mai 2017**

Promotion im Fachgebiet Sicherheit in der Informationstechnik im Fachbereich Informatik der Technischen Universität Darmstadt unter der Leitung von Prof. Dr. Michael Waidner.

## **2008 - 2010**

Masterstudium an der Norwegian University of Science and Technology und an der Technical University of Denmark.