LIMITING INFORMATION EXPOSURE IN MULTI-DOMAIN MONITORING SYSTEMS

BY

MIRKO MONTANARI

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Doctoral Committee:

Professor Roy H. Campbell, Chair
Professor Carl A. Gunter
Professor William H. Sanders
Professor Xinming Ou, Kansas State University

ABSTRACT

Security monitoring systems have been recognized as a fundamental component of security management, and they provide the fundamental building blocks of future reactive and autonomic systems that can automatically respond and adapt to changes in their environment. However, operating security monitoring systems in the complex environment of today's organizations is challenging. The complex structure of many organizations, the use of cloud computing, and the complexity of attacks require monitoring systems that can operate across the organization boundaries to integrate many types of information. However, when multiple security domains are involved, privacy and confidentiality problems create challenges in integrating events across systems. Situational awareness can be impacted, and so can be the ability of future systems to adapt to their environment.

Our thesis is that *the explicit definition of policies enables the design of multi-domain monitoring systems that protect the confidentiality and the integrity of the monitoring data*. We focus on the problem of sharing discrete events across organizations for detecting violations of security policies. We identify several scenarios from real-word policies in which such a multi-domain sharing is necessary. We introduce a novel architecture for monitoring multi-domain systems, and we introduce two complementary approaches for reducing the amount of information to share to a value close to the theoretical minimum. Our results show that our approaches have adequate performance in many monitoring scenarios, and significantly reduces the amount of information to share. Finally, as security monitoring is a fundamental service in modern systems, we provide a security analysis of our architecture. We analyze the impact of attacks on the integrity, availability, and confidentiality of the monitoring data. We show that, in many cases, our monitoring system fails gracefully in case of attacks without the causing catastrophic security failures of centralized systems.

To my family

ACKNOWLEDGMENTS

This dissertation could have not be possible without the support of many people, both colleagues and friends. I would like to thank my advisor, Roy H. Campbell, who gave me directions and many great ideas. Also, I would like to thank my thesis committee members, Carl Gunter, Xinming (Simon) Ou, and William H. Sanders, who provided great feedback on my research and helped me focus on important research problems.

Moreover, I would like to thank all the people with whom I had the luck and privilege of collaborating during my time at the University of Illinois (in alphabetical order): Zahid Anwar, Rakesh B. Bobba, Ellick Chan, Lucas T. Cook, Riccardo Crepaldi, Derek Dagit, Alejandro Gutierrez, Jun Ho Hun, Kevin Larson, Mingyan Li, Krishna Sampigethaya, and Wucherl Yoo. All of them provided important contributions in developing the research direction and the solutions contained in this thesis. In addition, I am grateful to all the members of SRG for all the great discussions that we had in the lab about many research (not-so-much research) topics.

I would like to thank all the many friends that supported me and made the past several years fun. And finally, I would like to thank my parents and all my family for their love and support.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Thesis Statement: *The explicit definition of policies enables the design of multi-domain monitoring systems that protect the confidentiality of the monitoring data*

Security monitoring systems have been recognized as a fundamental component of security management: the National Institute of Standard and Technology (NIST) mandates the presence of security monitoring capabilities in government's and contractor's systems [1]; industry standards such as the Payment Card Industry Data Security Standard [2] mandate similar requirements for private organizations. Modern monitoring systems are complex distributed systems that integrate events coming from a variety of sources and correlate them to identify conditions that can negatively affect security. These monitoring systems are the fundamental building blocks of future reactive and autonomic systems that can automatically respond and adapt to changes in their environment.

However, the environment where such systems operate is changing rapidly. The complex structure of many organizations, the use of cloud computing, and the complexity of attacks requires monitoring systems that can operate across the organization boundaries to integrate many types of information. When multiple security domains are involved, privacy and confidentiality problems create challenges in integrating events across systems. Situational awareness can be impacted, and so can be the ability of future systems to adapt to their environment.

In this work, we analyze the problem of sharing information across domains for the purpose of monitoring the compliance of a system to a set of security policies. We show that current approaches for sharing data across organizations cannot support an important use case: the sharing of discrete

event data in dynamic multi-organization systems. While a large amount of work has been focusing on sharing and preserving the privacy of data sets through aggregation and anonymization [3, 4, 5], many of the data required for evaluating the security of an infrastructure is discrete and cannot be summarized or anonymized easily. Information such as configuration changes, failures of specific machines, or dependencies between services cannot be aggregated easily: one single event could make the system operate in an insecure state. Current techniques for performing such an analysis rely on trusted central servers where information is concentrated. However, the challenges for integrating information from multiple cloud providers, users, and service providers into one location cannot be underestimated.

We analyze the fundamental characteristics of the problem to identify a theoretical minimum amount of information that needs to be exchange between domains for ensuring the identification of all policy violations. We show the tradeoffs in reaching such a minimum amount of information sharing across multiple organizations. We introduce two complementary approaches to the problem that minimize information sharing. Both approaches take advantage of general characteristics of infrastructure monitoring systems to reduce significantly the amount of information shared, while guaranteeing that all policy violations are still detected. We implement and evaluate such approaches by developing several techniques in the context of a novel distributed monitoring system.

The first approach uses information about the *observability* of the system in the determination the events to exchange. If information about a portion of the system can be observed completely within a domain, all policy violations that are contained within such a portion do not require information from other systems to be identified. We extend this intuition to develop a set of policy rewrite algorithms that distribute part of the event processing to each domain, and we prove the algorithm correct. Local violations are detected with no event exchange, while global violations are detected through the use of a distributed protocol that imports within the domain the information necessary for validating them. Our experiments show significant reduction in the amount of information exchanged in the case of a SNMP monitoring system.

The second approach explicitly represents the system as a set of interacting resources (e.g., computer systems, programs, users). As policies describe a

relation between the states of a set of resources, we identify violations by exchanging information only among monitoring systems managing interacting resources. Events are shared only if the state of the local resource potentially contributes to a global violation. We introduce a policy-rewrite algorithm, and we design information exchange protocols based on it. Our experiments show that such a technique significantly reduces the information exchange, and that the use of secure two-party computation [6] reduces even further the exchange of events to the theoretical minimum for simple policies, and maintains the exchange of events limited even for longer policies.

Additionally, our experiments show that our techniques are highly scalable, as computation is distributed across multiple servers. Moreover, we show that our architecture provides advantages in the security of the monitoring system itself. We provide evidence of the applicability of our techniques for designing intrusion-tolerant policy-based monitoring systems that are able to distribute information and computation across several hosts to reduce the effects of host compromises and of stolen credentials.

The rest of this thesis is structured as follows. Chapter 2 provides an overview of the related work in the area of monitoring, multi-domain monitoring, and securing monitoring systems. Chapter 3 introduces our monitoring architecture, our language for defining policies, and provides a proof of the minimal information sharing between domains. Additionally, the chapter includes several examples of using our framework for representing policies used in regulatory documents. Chapter 4 introduces our techniques based on observability and locality. We provide an algorithm for rewriting policies so that locality of the information is taken into account to reduce the amount of information shared, and we provide a protocol for exchanging information with another organizations to ensure that the detection of violations that cannot be detected only with local information. Chapter 5 introduces our techniques based on the resources. We introduce two policy-rewriting algorithms and protocols for selecting which information is exchanged between organizations. Chapter 6 provides the experimental evaluation of our techniques and shows how the different techniques affect the amount of event exchanged. Chapter 7 provides a discussion of the security of the architecture. We provide an experimental evaluation of specific security properties to evaluate the impact of attacks on the infrastructure. Finally, Chapter 8 summarizes our contributions and provides several avenues for future work.

CHAPTER 2

BACKGROUND AND RELATED WORK

This chapter provides an overview of the concept of security policy that we use throughout this work, and provides an overview of the related work in the area of monitoring for compliance with policies across multiple organizations.

2.1 Security Policies and Monitoring Rules

The management of the security of large infrastructure systems often relies on defining "policies." Policies are rules that specify high-level security requirements and are used for selecting the proper states and configurations of systems. In the past few years there has been an increasing interest for the development of policies that apply to entire classes of industries. Regulatory entities and industries introduced classes of policies such as PCI-DSS [2] for credit card industries, NERC CIP [7] for power grid systems. Several of the policies specified in NERC CIP or PCI-DSS relate to network, OS, and application configurations. We call such policies "infrastructure security policies," and their goal is to specify a minimum level of security to which all systems need to comply. Network administrators can monitor the compliance of their systems to these policies by specifying *monitoring rules*. These rules validate the configuration of security devices, of server applications, and of end-host computer systems; they detect when infrastructure security policies are violated. For example, a completely automated monitoring for compliance with one of NERC CIP policy requires defining rules that detect when machines are used by the critical devices that control the power grid, and that ensure that such machines are placed within a protected electronic perimeter (e.g., firewalls).

Maintaining and operating a system in compliance with such security requirements is a critical issue in any large system. Even when systems are

designed and built to be compliant to such security requirements, changes in the configuration of the system, or changes in the security requirements themselves make necessary a continuous process of monitoring to ensure that the system is still operating correctly [8]. To enable automated systems to monitor and to detect violations of the security requirements, monitoring systems collect information about the systems' operation from a variety of information sources and integrate them to reconstruct a partial view of the current system's state. For example, software such as Splunk [9], Bro [10], and SEC [11] integrate information collected by logs, by network packet analysis, or by SNMP systems.

The information collected by the monitoring systems can be represented as sequences of events. Events indicate an "interesting" change in the state of the system. Examples of events are the running of a new program, the log-in of a user, or the detection of a potential attack by an IDS. Infrastructure security policies check for the co-occurrence of events to identify when the system is operating in an undesirable state. For example, the co-occurrence of an IDS event indicating the detection of an exploit packet and of a vulnerability-scanner event indicating the presence of a software vulnerable to such an exploit signifies the possible compromise of a device. The events used by these systems are generated by a large number of devices using a variety of sources such as SNMP data, intrusion detection systems (IDS), and log-analysis tools.

Infrastructure security policies could define security requirements in a way which is orthogonal to the methods used for enforcing security and they can be used for providing an audit trace. For example, the NERC CIP policy requirement above might be implemented using firewall systems. A monitoring rule might monitor for events that identify critical systems and for events that indicate the presence of connections from outside the electronic perimeter. In the rest of the paper we use the terms monitoring rule and policy interchangeably to indicate an event-correlation rule.

2.2 Multi-Domain Systems

Multi-domain systems are common in today's computing infrastructure. The services offered by cloud computing systems, critical infrastructure systems,

or healthcare systems are often provided through the interaction of a computing infrastructure managed by several organizations. For example, in a hybrid cloud environment, services provided by the infrastructure of an organization are integrated with services managed by a cloud provider. In intercloud systems [12], or in systems based on cloud brokers [13], multiple cloud services are integrated to provide a service to cloud users. Such services are provided by a variety of cloud providers, and their selection might depend on dynamic conditions not known beforehand. Critical infrastructure systems such as the US power grid is composed of several systems that interact to coordinate the production and distribution of electric power. In airport systems, providing advanced network services to aircrafts requires an interaction between infrastructure managed by airport owners, airlines, and multiple maintenance contractors.

In such settings, it is common to have multiple independent monitoring systems acquire information about the infrastructure. For example, in the case of hybrid clouds, monitoring systems in the private infrastructure acquire application-level information from software running on the local infrastructure and on cloud instances. Monitoring servers managed by the cloud provider acquire information about the physical location of virtual machine instances, the colocation of virtual machines with other customers, and the load of the infrastructure. In these settings, conditions to detect violations can be complex, and analyzing each organization's information independently is not sufficient to detect violations. However, sharing monitoring information outside an organization is often undesirable. Information about the infrastructure configurations provides details about security postures or information about the internals of an organization to competitors.

Because of the tight interaction between the infrastructure of the organizations, many security problems arise because of undesirable interactions between devices and configurations managed by different organizations. Detecting such situations requires integrating and analyzing such information across the multiple organizations. However, the boundary of each domain poses an artificial barrier to the flow of information about the system's state that complicates the process of information integration.

Many regulatory documents are designed to avoid the problem of information sharing. Different security requirements are created for each organization involved, and each system independently monitors its own infrastructure for

compliance. While such a solution is feasible when considering simple security requirements, the validation of complex policies requires checking for conditions spanning multiple domains.

A significant amount of work has been performed in the development of techniques for monitoring policy compliance and for sharing data across organization boundaries. The next section provides an overview of the state of the art in such areas, and shows how new techniques are needed to permit the detecting of inter-domain violations of invariants while limiting the amount of information crossing the organization boundaries.

2.3 Related Work

The problems that we address in this thesis have been analyzed in different contexts. This section provides a description of the work relevant to our efforts.

In Section 2.3.1, we analyze related work in the area of policy-based monitoring. We show that policy-based monitoring has a significant application in research and industry. Such related work strengthens the motivation for our research. Our research enables the use of policy-based monitoring systems in contexts where unrestricted information sharing is unacceptable. Section 2.3.2 outlines the advancements in the topic of securely sharing data across different domains. We show that the current solutions are not suited for policy-based monitoring in dynamic environments. Section 2.3.3 outlines the progress in creating secure monitoring systems. We show the problems in current solutions, and we outline the advantages of our monitoring architecture. Section 2.3.4 describes the similarity of our work with the area of distributed debugging and outlines the differences between our approach and previous approaches in such an area.

2.3.1 Policy-based Monitoring

Policies have a long history in computer science. Work on policy has been started in the '70 in the context of time-sharing mainframes. Initially, policy focused on security and were limited in defining rules according to which access control must be regulated [14]. Later, the use of policy has been

extended to network and to distributed system management.

The Bell-LaPadula model [15] is one of the first formalizations of security policy. Such a security model focused on confidentiality policies and was introduced in 1973. In 1977, Biba [16] introduced an integrity policy that complements confidentiality policies and describes how the validity of the data is protected from modifications. Other types of policies followed, such as the Chinese-wall policy [17] which has been introduced for abstracting British financial regulations. More recently, Role-based Access Control (RBAC) [18] has gained wide acceptance in the field of access control, as it can lower the cost and the complexity of securing large systems through a simpler and more management definition of access control policies [19].

While the original literature on access control focused on simple systems or single organizations, more recent work extended the approaches to systems composed of multiple domains. Earliest work focused on extending the concepts of RBAC to multi-domain systems. Hayton et al. [20] introduces a system that allows users to obtain from multiple domains certificates indicating their roles and permissions, which can then used for obtaining access to restricted resources. KAoS [21] is a distributed system for enforcing access control to a distributed set of objects. It is based on ontology for the definition of policies, and supports the use of reasoning in policies.

Approaches to access control in multi-organizations systems have been proposed in the context of event-based systems. In event-based systems, events are pushed to their destination often without an explicit request from the user. While in the traditional model of access control each request can be verified to ensure that the user is authorized to access the data, in event-based systems events need to be filtered after creation to ensure that they are not published outside the domain and received by unauthorized users. Singh et al. [22] introduce a system defining explicit confidentiality policies based on classes of events. Through such a policy, event classes containing critical data can be stopped from being delivered outside the domain. In a similar manner, Evans et al. [23] propose a solution in which events are tagged with labels, and those labels are used to enforce access control.

More recently, the use of policy extended beyond pure access control, and policies have been used for simplifying the general management of systems. In particular, Robinson et al. [24] defines management policies as a set of rules for achieving the scalable management of a distributed system. More

recently, the DMTF Web Based Enterprise Management (WBEM) framework [25] introduces a unified access to the configuration of each device and provide a support for building policies that relate to the configuration of systems. Netquery [26] provides a mechanism for integrating such information across multiple devices and across multiple organizations using explicit access control policies.

Policy-based approaches are found in most recent security regulations used in different sectors of industry. For example, the Payment Card Industry Data Security Council [27] defines a set of security regulations for organizations handling credit card data from the major credit card companies. In addition to access control policies, such regulations includes a set of conditions on the configuration of systems that ensure a minimum level of security in an infrastructure. Other regulations follow the same structure, but apply to different industry sectors. The National Institute of Standard and Technology (NIST) defines a set of regulations for compliance with the Federal information Security Management Act (FISMA) [1] that specify valid configurations for computer systems in organizations that interact with the US Federal government. The North America Electric Reliability Corporation (NERC) defines a set of standard called Critical Infrastructure Protection (CIP) [7] that specify minimum security configurations for systems interacting with the US power grid.

In this work we focus on policy-based approaches for management and we address the problem of sharing data across domains. In our context, the sharing of data is a consequence of the need of detecting policy violations and it is not controlled by an explicit access control policy. Data sharing should be limited to the information necessary for the detection of invariant violations. Static policies which definition is based on the recipient of the data and on the type of the information are limited in addressing our problem: very restrictive policies might not share events required for detecting a global violation, while open data-sharing policies would reveal unnecessarily a large amount of information to the other party. This thesis analyses the tradeoff between sharing and detection of invariant violations. Our solutions implicitly define dynamic sharing policies that depend on the state of the system. The information to share changes depending on the state of the local system, and on the state of other connected systems.

Other work takes a centralized approach to the problem of ensuring com-

pliance. Systems have been proposed for performing specific security assessments and can be used to monitor compliance with regulations. Network scanners and security assessment tools such as TVA [28], or MulVAL [29] acquire information about the configuration of the system by using port scans or direct access to hosts. ConfigAssure [30] takes a top-down approach and synthesizes network configurations from high-level specifications. Such an approach is applicable within a single organization. However, when multiple organizations are involved, the approach requires a significant loss of independence in the operations of each domain, as configuration managed is centralized.

The fundamental problem of the centralized approach is that it relies on central servers for performing the analysis. When multiple domains are interacting, the validation of complex inter-domain policies requires integrating all the information about the systems' operations in a single trusted entity. Such an action leads to several problems. First, finding a trusted central entity that can receive complete access to the internal security configurations of all the involved organizations is challenging. Second, such a centralization creates important security risks: a single compromise can open multiple systems to attack. Third, the central system can become a bottleneck for the analysis and a single point of failure.

Our distributed architecture does not require centralizing the data. Each organization independently operates its own monitoring system and exchanges information with other organizations when needed. We aim at reducing such an exchange of information to a minimum level.

2.3.2 Anonymization and Log Protection

Collaboration between organizations for detecting attacks and other security problems has been an important topic in research for several years. The interdependencies between systems that we find in today's cloud computing increase the need of such a collaboration. However, sharing information presents several security problems. Monitoring data contain valuable information about the organization's computing infrastructure. Such information can be used by attackers to find possible attack methods; and can be used by competitors to get valuable insight in the internal operations of the orga-

nization. For this reason, a significant amount of work has been focusing on reducing information sharing, while still permitting the detection of complex event patterns.

Outside the context of configuration management, several techniques focus on hiding log information through anonymization [3, 4, 5, 31, 32, 33]. Lincoln et al. [5], in particular, introduce a technique for removing critical data from network traces. SEPIA [34] provides a threshold-based mechanism for sharing aggregated data about network traffic. Denker et al. [35] use a selective downgrade of GPS data for sharing location data. However, such techniques generally apply to numeric information and the summarization is strongly dependent on the semantic of the information. Our techniques focus on discrete data that cannot be summarized without losing the ability of detecting policy violations correctly, such as configuration changes, failures of specific machines, or vulnerability information.

Huh and Lyle [36] discuss a trusted computing way to enable "blind log analysis," allowing different organizations to freely share raw log data with the guarantees that their raw data will not be revealed to other organizations. A trustworthy log reconciliation service is attested and verified, providing assurance that all the log reconciliation and analysis is performed blindly inside a protected virtual machine, and that only the fully processed, privacy-preserving analysis results are made available to other organizations. Huh and Martin [37] extend this work and propose the concept of a "blind analysis server", which allows privileged data analysis to be performed securely and privately through a remote server. In our approach, information leaves the security domain only if necessary. We reduce the reliance on external software for protecting the confidentiality of data and we do not require the entire infrastructure to support remote attestation [38]. However, if such an infrastructure exists, we can validate remote servers and increase the confidence that information does not leave the domain unnecessarily.

Other techniques focus on integrating data in a central server for analysis. Australias Commonwealth Scientic and Industrial Research Organisation (CSIRO) has developed a Privacy-Preserving Analytics (PPA) software for analyzing sensitive healthcare data with confidentiality guarantees [39]. PPA performs analysis on the original raw data but modifies the output delivered to researchers to ensure that no individual unit record is disclosed. This is achieved by removing any directly identifying information and deductive

values that can be matched to external databases. Closely related to our work, Lee et al. [40] introduce a framework that allows a group of organizations to share encrypted logs with a central auditor. The auditor analyzes the encrypted logs and detects attacks or other policy violations. Our work improves on such approaches by removing the need to store centrally the logs collected from all organizations. While having a central authority is feasible in certain situations, in cloud and in cloud-of-clouds systems, organizations can integrate resources across multiple providers and provide them to different clients at runtime. Having all entities involved push out their logs to a single central location is challenging. Our approach uses a distributed mechanism for correlating data, and security domains interact directly only when they require information about specific external resources.

2.3.3 Protecting the Monitoring Infrastructure

The increased protection provided by our system is based on the principles of intrusion tolerance [41]. We exploit the distributed nature of the problem of event-correlation to provide a specific intrusion-tolerant solution. Other approaches to the problem of securing monitoring systems have been proposed over time. This section summarizes a few important milestones.

Monitoring is a widespread service in modern systems. Most monitoring systems provide some limited protection of log data confidentiality. A basic protection of confidentiality is provided by protecting data-in-transit. For example, *syslog-ng* [42] uses TLS to transmit encrypted log data from the devices and the monitoring servers. However, TLS encrypts data in-transit and does not provide any protection once the log has been stored on disk or in memory: if the monitoring server is compromised, the attacker has access to all past logs and can capture future data.

More advanced solutions provide protection of the data-at-rest by creating encrypted and tamper-proof audit logs that can be accessed only by authorized users. One of the earliest mechanisms has been introduced by Schneier et al. [43]. Their approach uses one-way hash chains to protect the log files from modifications. Additionally, logs are encrypted to protect them from unauthorized access. Other solutions (e.g., Ma et al. [44]) extend such an approach to provide additional integrity protection. While these approaches

are useful for protecting the integrity of the event logs and can be used in conjunction with our algorithm to provide trusted audit logs, their confidentiality protection is not suited for our scenario: as event correlation requires performing processing on the data, events need to be accessible to the monitoring server. Attackers compromising the server would have access to such data.

The Intrusion Detection System Bro [10] provides a distributed mechanism for performing event correlation. Communication between Bro nodes is performed on top of SSL to protect data-in-transit. Correlation between events is performed by programming policies using a pub/sub mechanism. For example, a distributed IDS cluster built on top of Bro has been presented by Vallentin et al. [45]. They use a flow-based hashing for distributing the packet-processing load across the instances. Inter-flow correlation is performed using the pub/sub mechanism. While the pub/sub mechanism provides flexibility in specifying policies and in defining their evaluation, it provides no guarantees that information about the system is distributed across nodes. It is up to the programmer to evaluate policies without creating such an aggregation of information. The algorithms we present in this work provide a mechanism for selecting automatically the events that each monitoring server should receive and send for validating policies. Our algorithm ensures that potential policy violations are detected and that information about the system is distributed across a large number of hosts.

Other work focuses on protecting the monitoring system itself from compromises. For example, recently several secure monitoring solutions have been using Virtual Machine Introspection (VMI) for protecting the monitoring software from compromises. They run the monitoring software in a separated VM co-located with the host to monitor (e.g., Livewire [46]) and they access information by analyzing memory and disk data without the OS mediation. The security of these systems relies on the fact that compromising the monitoring VM is harder than compromising other VMs: the monitoring VM runs a small amount of software and, hence, exposes a small attack surface. However, while this assumption holds if the monitoring VM is used only for acquiring events from a particular system, it does not hold in the processing servers that correlate events across entire organizations. Such hosts need to be accessible through the network to allow devices to send events to them, and they need to run a substantial amount of software for

validating policies and for providing network administrators access to data. Our architecture reduces the consequences of compromises of such servers.

2.3.4 Other Related Areas

Distributed system debugging is another area that overlaps with policy-based monitoring, even if it presents significant differences. The goal of distributed debugging is to validate that an execution of a distributed application is satisfying a set of invariants defined on the program. A wide set of techniques have been developed for enabling such debugging to scale [47, 48], and previous work used logic languages for expressing conditions on the system [49]. To the best of our knowledge, none of the distributed debugging techniques include provisions for limiting the sharing of information across domains, as the information collected is assumed to be free of security-critical data. However, even in distributed debugging part of the state of the application could contain security-relevant events. In such cases, the use of our techniques could be beneficial.

CHAPTER 3

MONITORING POLICY AND ARCHITECTURE

This chapter introduces the architecture and the policy language of our monitoring system. We take several examples of policies from regulatory documents and we show their representation in our framework. We analyze the theoretical limits to reducing the sharing of information. We show that our architecture builds on top of monitoring systems independently operated within each security domain. Organizations use the local monitoring system to identify violations of policies that are completely internal to their own infrastructure, and use inter-domain communication to detect problems that span across multiple security domains.

The rest of the chapter is structure as follows. Section 3.1 provides the overview of our distributed monitoring architecture. Section 3.2 describes our assumptions. Section 3.3 describes our logic model for representing events and states of the system. Sections 3.6, 3.7, 3.8 describe how policies in industrial domains are modeled in our framework. Section 3.5 describes our results in the determination of a minimal need-to-know set of events between two organizations.

3.1 Architecture Overview

Monitoring of complex systems for configuration errors, security breaches, or regulation compliance requires a large amount of information to be collected (usually in the form of audit logs) and analyzed. In distributed environments like clouds or cloud-of-clouds [12], this monitoring may require logs to be shared across multiple security domains to detect particular security events. However, some of those logs might contain sensitive information about customers or might have commercial value. Without the necessary confidentiality and privacy guarantees, most organizations will be reluctant

to share such privileged logs with others.

As anonymization is challenging to use in our scenario, a simple solution would be integrating information in servers common among the organizations. However, such a solution faces several organizational challenges. As each organization prefers to maintain its information private, the information integration needs to occur in a third-party entity trusted by all the organizations. Moreover, because such multi-domain systems can be created dynamically by the addition of interactions through public APIs (e.g., in cloud computing), such third-party entity would need to be trusted by any new organization joining the system at runtime.

To address such a problem, our distributed architecture relies on monitoring servers operated independently by each organization. Each server acquires information about the state of the infrastructure and detects independently problems in the local infrastructure. The servers exchange information to detect policy violations caused by the interactions of events generated by different parts of the infrastructure.

Each monitoring server acquires information about the infrastructure state through a wide range of information sources. For example, network management systems and system logs acquire information about configurations of devices and their operational state; IDS acquire information about communications between systems and possible attacks; and application logs provide semantically rich information about the software state.

For example, a simple deployment of our infrastructure includes an enterprise system with a monitoring server acquiring information about the applications running on its own private infrastructure and on cloud instances; and a cloud provider with multiple monitoring servers acquiring information about the physical location of virtual instances, the co-location of virtual machines with other customers, and the load of the infrastructure. Events carrying such information are already aggregated for standard system management and monitoring purposes, so we do not expect our system to require additional monitoring.

On top of such an architecture, administrators define policies or "invariants" that identify undesirable conditions of the operational state. Our distributed algorithms guide the interaction between the monitoring servers placed within each domain to verify if the current operational condition violate one of the policies.

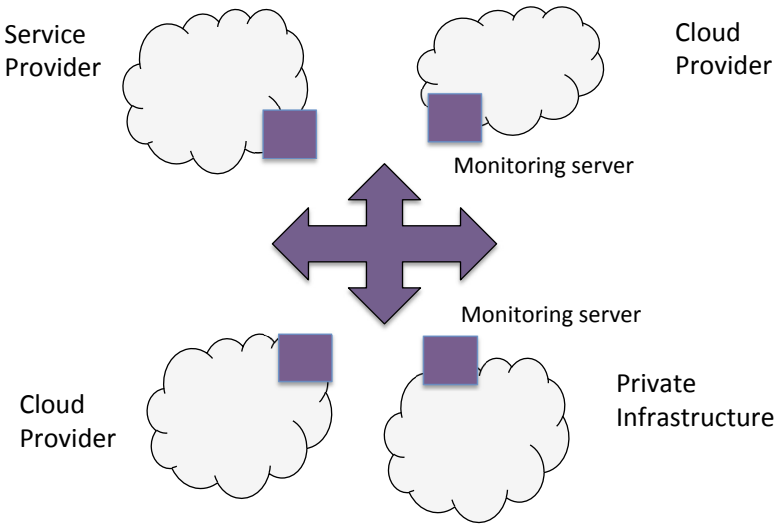

Figure 3.1: Architecture of our monitoring system. Multiple monitoring servers are placed in different security domains. Servers communicate to detect violations of policies.

3.2 Assumptions

Our architecture and our algorithms rely on assumptions made to focus our description on fundamental issues. We provide a list of our assumptions below. For each assumption, we outline mitigation techniques.

1. We assume that inter-domain policies are not confidential and that are shared among the domains, and that each monitoring server maintains a copy of them. The distribution of new policies is performed in parallel to the monitoring process through an external mechanism. Policy files are signed and we rely on a Public Key infrastructure to ensure their integrity. Such an assumption is verified when policies are mandated from agencies such as the NIST, PCI-DSS, or NERC CIP, or when knowledge of policies themselves do not provide competitive advantage to other organizations.

2. We assume that the events generated by the different monitoring systems can be represented using a common ontology of events. While different organizations can generate events with different characteristics, the communication of information about systems should rely on a common and shared ontology of events and resources. Previous work analyzed methods for reaching such a common representation [50], and

our system can take advantage of such research.

3. We assume that organizations behave according to the protocols and do not provide false information about their system. Different attack models, their implication, and mitigation techniques are described in Chapter 7.

4. We assume that clocks are synchronized so that event timestamps are comparable across organizations. The use of Network Time Protocol (NTP) permits to synchronize clocks within tens of milliseconds [51]. In our experience, infrastructure security policies specified in PCI-DSS or FISMA do not require strict synchronization between events.

3.3 Event Representation

Data sources within each domain generate events to provide information about the state of the system. Examples of events are information about changes in the configurations of systems, detection of network flows, or failures of specific machines. We use Datalog with negation as the common language for representing event data and policies. Datalog with negation is able to represent policies defined over a large class of relational calculus formulas [52]. These formulas can contain existential quantifiers, universal quantifiers, conjunctions, disjunctions, and negations. Policies might need to be expressed using multiple rules. In our analysis of the policies currently available in standard document, such language has been sufficient for representing all policies.

We use two representations of the data: an *event-based* representation and a *state-based* representation. Using the event-based representation, security administrators define sequences of events that cause a violation of invariants. Events are expressed using the Resource Definition Framework (RDF), and the temporal relation between events is expressed using interval logic. The state-based representation provides a simpler alternative for expressing constraints. As many of the policies defined in regulations such as PCI-DSS and NERC CIP specify directly state of the system that violate the policy, in the state-based representation we express constraints directly on a reconstructed partial state of the system. For example, we can easily express policies such

as "vulnerable systems should not run critical services" without having to represent explicitly the events involved and their relations.

Table 3.1 provides an example of a policy, and includes a list of the events or states which are involved in the detection of a violation. The next sections show how such a policy can be encoded using an event-based representation and a state-based representation.

3.3.1 Event-based Model

Using the event-based representation, each piece of information generated by information sources is directly represented as an individual event. Such events are collected and analyzed in the local monitoring servers. Using the event-based representation, policies express sequences of events satisfying a set of conditions. Such a representation permits great flexibility in representing temporal conditions of the systems.

An event is characterized by an unique ID and by a set of parameters (e.g., event type, event source, and event-specific data). We represent each event as multiple logic statements. Without loss of generality, we use the Resource Definition Framework (RDF). In RDF, each statement is a tuple $(id, property, value)$ composed of three parts. The first part is an event ID, which identifies uniquely the event throughout the system. The second part indicates the name of an event *property* and represents the type of information provided by the statement (e.g., the property `instance` indicates id of an virtual instance). The last part contains the value for the given

Policy example	**Events**	**Source**	**Res**	**Description**
Not run a critical service on a physical server which is sending malicious traffic	`criticalService`	Private Infrastructure	P, I	Critical service P is running on a VM instance I
	`instanceAssigned`	Cloud Provider	I, S	VM Instance I launched on machine S
	`badTraffic`	Cloud Provider	S	Malicious traffic detected from S

Table 3.1: Example of a multi-domain policy and events required for the detection of a violation. For each event, we list its source and the information it carries.

property. An event is composed of multiple statements having the same event ID. In our example, we represent an event with ID e_1 of type `criticalService` providing information that a VM instance identified with m is running a program p as follows:

$$\begin{aligned} &1 : (e_1, \text{type}, \texttt{criticalService}), (e_1, \text{pname}, p), (e_1, \text{instance}, m), \\ &2 : (e_1, \text{startTime}, t_s), (e_1, \text{endTime}, t_e), \end{aligned} \tag{3.1}$$

where `criticalService` is the type of the event; and the rest of the statements on line 1 provide information about the content of the event. The statements on line 2 indicate start time and end time of the event.

A policy identifies a sequence of events by expressing conditions over the logic statements in each event. We represent a policy using a Datalog rule. The body of the rule is composed of two sets of conditions: *event conditions* and *temporal condition*. Event conditions specify a conjunction of event properties. We express that two properties need to assume the same value for the policy to be satisfied by using the same variable name in both conditions. Temporal conditions express the temporal relations between start and end times of events. We use seven constraints (and their negation) to represent all possible temporal conditions between events [53]. The conditions are summarized in Table 3.2. For example, we can specify that an event e_1 "takes place before" e_2, or that an event e_1 "overlaps with" e_2. Such temporal conditions can be related using boolean conjunctions, and negations (disjunction of conditions can be represented by creating multiple rules). The head of the rule is a conjunction of logic facts that are created when the conditions of the rule body are satisfied. We use uppercase characters to identify variables and lowercase characters represent constants and ground terms. The verification of the policies is performed using a RETE-based reasoning engine [54] enhanced with support for interval logic [55].

As an example of a policy, we consider an event indicating that a physical server running the instance is receiving malicious traffic. The event contains the id of the physical server and the time. We represent the event as follows.

$$\begin{aligned} &(e_2, \text{type}, \texttt{badTraffic}), (e_2, \text{server}, ps), \\ &(e_2, \text{instance}, m_1), (e_2, \text{instance}, m_2), (e_2, \text{startTime}, t_{s1}) \end{aligned} \tag{3.2}$$

The policy of our example defines relations between two events. First, we define an equality relation between the value of the property `instance`

Constraint	Description
precedes	$x^+ < y^-$
meets	$x^+ == y^-$
overlaps	$x^- < y^- < x^+, x^+ < y^+$
during	$x^- > y^-, x^+ < y^+$
starts	$x^- == y^-, x^+ < y^+$
finishes	$x^+ == y^+, x^- > y^-$
equals	$x^- == y^-, x^+ == y^+$

Figure 3.2: Temporal policy constraints. x^- the starting time of an event x, and x^+ is its end time.

$1 : (E_1, \text{type}, \texttt{criticalService}),$
$2 : (E_1, \text{instance}, I), (E_1, \text{pname}, P),$
$3 : (E_2, \text{type}, \texttt{instanceAssigned}),$
$4 : (E_2, \text{instance}, I), (E_2, \text{server}, S),$
$5 : (E_3, \text{type}, \texttt{badTraffic}),$
$6 : (E_3, \text{server}, S),$
$7 : [E_1 \text{during} E_2] \wedge$
$8 : ([E_2 \text{overlaps} E_3] \vee [E_2 \text{during} E_3])$
$9 : \rightarrow (v_1, \text{violation}, I)$

Figure 3.3: Policy requiring to not run a critical service on a physical server which is receiving malicious traffic.

by setting it to the same variable name, M. Second, we define temporal conditions by using the condition within square brackets which specifies that the two events should `overlap` (i.e., the start time of E_1 is less than the start time of E_2, and E_1 ends after the start of E_2 but before the end of E_2). If two events satisfy both the event and the temporal conditions, the policy is violated. Fig. 3.3 shows the formal representation of the example policy of Table 3.1. The policy uses the same variable names in the values of the properties `instance` and `server` to define an equality relation between such properties in events E_1, E_2 and E_2, E_3. Temporal conditions are expressed within square brackets, as shown in lines 7-8. The policy is violated if three events satisfy all conditions.

Each monitoring system collects events into a local knowledge base KB for detecting policy violations. A violation v is detected if $KB \vdash v$. To avoid unbounded growth in the knowledge base, events are removed from it when, according to the constraints, they do not contribute to the detection of new violations [55]. We require rules to respect a set of conditions typical of Datalog¬ programs [56]. First, we require rules to be *safe-range*: variables appearing in the body or head of a rule need to appear at least once in a non-negated statement pattern in the body of the rule. Second, negation needs to be stratified: such a constraint imposes certain restrictions on the use of negation with recursion. Even with these constraints, the language is still able of representing complex policies.

3.3.2 State-based Model

The analysis of the policies contained in regulatory documents such as PCI-DSS [27] shows that many security constraints can be expressed as conditions on the state of the system at any single point in time. For example, a policy might require critical computer systems to be deployed in network protected by firewalls; another policy might require that instances of virtual machine running security-critical services to be running on approved data centers. If the state of the system at any time violates such conditions, we have a violation of the policy. While it is possible to express these policies using events and temporal conditions, the state-based model provides a simpler way to represent and to check for compliance to such rules.

The state-based representation is based on expressing conditions on the reconstruction of a partial view of the system's state. The state of the system is represented as a knowledge base (KB) of Datalog "facts." Events represent changes in such a KB. A policy is a Datalog rule defined over on the KB. Each condition of the Datalog rule is an *event pattern*. For example, we can represent the policy in Table 3.1 using three states. We indicate that a critical service P is running on a VM machine I at the current time using a Datalog fact `criticalService(P, I)`. We indicate that an instance I has been launched on a physical server S using `instanceAssigned(I, S)`. We indicate that a physical server S is receiving bad traffic using the fact `badTraffic(S)`.

We represent the policy using the following rule.

$$\begin{aligned} &\texttt{criticalService}(P, I), \texttt{instanceAssigned}(I, S), \\ &\texttt{badTraffic}(S) \rightarrow \mathit{violation}(I). \end{aligned} \tag{3.3}$$

Events add or remove statements from the knowledge base to indicate changes in the state of the system. We associate two events to each statement: a start event, indicating that the given fact becomes true in the system; and a end event, indicating that the fact is now not true anymore and should be removed from the knowledge base. Such an handling of events and the restriction in the specification of temporal constraints (i.e., it represent a current state) makes the specification of state-based policies easier.

A policy compliance monitoring system receives events from the information sources and applies the changes to a local knowledge base KB. A violation v exists when $KB \vdash v$.

3.4 Multi-Organization Representation

The event-based and the state-based representations maintain the knowledge about the system in a knowledge base K. While the two representation model facts and transitions in different ways, the different is just syntactical. Any set of facts in the state-based presentation can be represented using the event-based representation. Rules play a fundamental rule for detecting the presence of violations. Rules define explicitly new facts about the system that can be inferred from the given data. Such new facts can either define new complex events or policy violations.

We define formally the concept of *violation* the ability to infer a "violation statement" from the current state of the system. More formally:

Definition 1. *Consider K^+ the fixpoint model of K with the rules R. Given a rule $r \in R$ which rule head is a statement v indicating a violation, we say that a* violation *v exists in a system if $K^+ \vdash v$. If v exists, we say that the infrastructure is* not compliant *to the policy r.*

When it is possible to infer the presence of a violation, we say that the infrastructure is not compliant to a given rule. Generally, a violation statement contains information useful for describing the type of violation and the entity involved. Administrators use information from the violation statement to address the problem identified by the monitoring system.

In a multi-organization scenario, each monitoring system maintains a knowledge base for the events collected locally. We focus on the case of two organizations, A and B, sharing their infrastructure. We indicate with K_A and K_B the sets of events collected by the monitoring system of each organization, and with $K = K_A \cup K_B$ the set of events collected by both systems. R is the common set of rules defining violations and complex events.

Applying reasoning independently in each organization KB leads to *incomplete* and *incorrect* results.

1. **Incomplete:** A violation that exits in the system cannot be detected. Formally, there exists a violation v such that $K \vdash v$, but neither $K_A \vdash v$ nor $K_B \vdash v$. An incomplete reasoning occurs, for example, if parts of the events causing violation v are stored in K_A, and the others are stored in K_B.

2. **Incorrect:** A violation that does not exist in the system is detected. Formally. there exists a violation v such that $K_A \vdash v$ or $K_B \vdash v$, but $K \nvdash v$. An incorrect reasoning occurs, for example, if the violation requires the non-existence of an event e. If the event is stored only on K_B, K_A might incorrectly assume that the event does not exists.

The lack of completeness and correctness are a consequence of the closed-world assumption used by Datalog reasoning. In the closed-world assumption, if an event e is not found in the local knowledge base, then such an event does not exists. However, in our case, the event can still exists and it can be stored in K_B. Our goal is to obtain a complete and correct reasoning while limiting the amount of information transferred across organizations.

3.5 Need-to-Know Events

We define the minimum amount of information that needs to be transferred between organizations to detect a violation v. We call such a set of events *need-to-know event set*. For two organizations, we formally define it as follows.

Definition 2. *Given an organization A and a violation v such that $K^+ \vdash v$, we define a set of facts $E \subset K_B$ as a* need-to-know *set when $K_A \cup E \vdash v$, and there is no set $E' \subset E$ such that $K_A \cup E' \vdash v$.*

The set of facts E contains the minimal amount of information that needs to be shared by organization B to allow the detection of a violation v. The set E can be empty if no additional information is needed to detect v (i.e., $K_A \vdash v$), or if there is no violation in the overall system. In other cases, the minimum amount of information depends on the policies and on the violation detected.

In our system, policies are known to all parties. Hence, once a system knows that a specific violation exists, it is possible to identify univocally the set of conditions occurred in the overall system that leads to the given violation. We show that, in such conditions, the minimum amount of information to share is the information that can be inferred from the knowledge of the existence of a violation. We prove such a result below.

Theorem 1. *We take an organization A and we consider a ground statement θh indicating a violation, so that $K^+ \vdash \theta h$. The statement θh is consequence of a rule $b_1, \ldots, b_n \rightarrow h$ and a substitution θ that satisfies the body of the rule. We assume that no other rule has a consequence the statement θh. We divide the statements in the body of the rule in two sets B_1 and B_2. The set B_1 contains statement patterns so that θB_1 refers to events local to A, while θB_2 refers to external events.*

We take a subset θ_h of the substitution θ by considering only variables that appear in the head of the rule h. The minimum need-to-know information for the organization A and the violation θh is $\theta_h B_2$.

Proof. We provide the theorem by contradiction. Let's assume that the set E that A needs to know for $K_A \cup E \vdash \theta_h h$ is a strict subset of $\theta_h B_2$. Hence, some statement contained in $\theta_h B_2$ is not known in $K_A \cup E$, but still θh is true.

By assumption, organization A is in charge of detecting the violation. Hence, A needs to know the presence of the violation in the system, which we represent as $\theta_v h$ (the head of the rule and the values of all the variables that appear in such a statement). In our system, we assume that the rules generating violations are known to all parties, hence A knows the rule $b_1, \ldots, b_n \rightarrow h$. Because $\theta_h h$, if true, could only be implied by the rule $b_1, \ldots, b_n$, we have that there must be a substitution $\theta' \supseteq \theta_h$ that satisfies the body of the rule. Based on such information, we infer that the events $\theta' B_1$ and $\theta' B_2$ are in $K_A \cup E$. Hence, we have a contradiction, as we assume that $K_A \cup E$ does not contain events matching $\theta_h B_2$. □

The set of facts E might contain only partial information from events, as often part of the information carried by an event is not needed for the detection of a violation. We can reduce information by removing the name of some resources mentioned in the event. An event relates n resources with each other under a specific relation. For example, the event $\texttt{runs}(host_1, pid_1, apache)$ relates the resource $host_1$ with the PID pid_1 and with the software *apache*. Often, we can detect a violation by knowing that a relation exists between a resource and an undefined resource with specific characteristics. For example, we can consider a rule $\texttt{violation}(P) \leftarrow \texttt{runs}(H, PID, P), \texttt{vulnerable}(P)$ and assume that A maintains a list of vulnerable programs, while B provides the actual services. If A knows $\texttt{vulnerable}(apache)$, the only information

missing from B is the knowledge about the existence of at least one host that runs such a service. As organization A is only interested in knowing which vulnerable programs are run and not the actual name of the host running them (i.e., the host name is not part of the parameters of `violation`), we can remove the host name from the events passed to A.

In first order logic we represent the existence of a host running *apache* using existential quantification as in $\exists H, P : runs(H, P, apache)$. This statement expresses the fact that `runs` is true for some value of H, P. Multiple pieces of partial information can be related to each other to express that two events are related to the same resource. For example, we can express that some critical server runs apache using $\exists H, P : critical(H) \wedge runs(H, P, apache)$. We define *knowledge units* to be existentially quantified expressions in the form $\exists V_1, \ldots, V_n : e_1 \wedge \ldots \wedge e_k$. In our Datalog framework, we represent knowledge units using Skolemization: we substitute the existential quantified variables with unique constants. As Skolemization preserves satisfiability, if a violation exists, we can find it in the Skolemized version of the system.

3.6 Policy Example: PCI-DSS

PCI-DSS v2.0 has been introduced by the major credit card companies to raise the level of security of merchants working with cardholder data. Such a regulation is composed of six control objectives, with each objective divided into groups of high-level requirements, as show in Table 3.2. For each high-level requirement, the document specifies in more detail policies that should be respected in the configuration and operation of the infrastructure.

First, we analyze the different requirements to identify the ones containing infrastructure security policies. We find that such requirements are contained in the control objectives "Build and Maintain a Secure Network" (Requirements 1 and 2) and "Maintain a Vulnerability Management Program" (requirements 5 and 6). Additionally, a limited number of requirements in "Implement Strong Access Control Measures" applies to configurations of the network. The remaining policies cover requirements about access control (Protect Cardholder Data, requirements 3 and 4; and part of "Implement Strong Access Control Measures", requirements 7, 8, and 9) and they can be monitored through a proper access control mechanism. Additionally, the

Table 3.2: Overview of PCI-DSS: Control objectives and requirements

Build and Maintain a Secure Network
Requirement 1: Install and maintain a firewall configuration to protect cardholder data
Requirement 2: Do not use vendor-supplied defaults for system passwords and other security parameters
Protect Cardholder Data
Requirement 3: Protect stored cardholder data
Requirement 4: Encrypt Transmission of cardholder data across open, public networks
Maintain a Vulnerability Management Program
Requirement 5: Use and regularly update anti-virus software or programs
Requirement 6: Develop and maintain secure systems and applications
Implement Strong Access Control Measures
Requirement 7: Restrict access to cardholder data by business to need to know
Requirement 8: Assign a unique ID to each person with computer access
Requirement 9: Restrict physical access to cardholder data
Regularly Monitor and Test Networks
Requirement 10: Track and monitor all access to network resources and cardholder data
Requirement 11: Regularly test security systems and processes
Maintain an Information Security Policy
Requirement 12: Maintain a policy that addresses information security for all personnel

control objectives "Regularly Monitor and Test Networks" and "Maintain an Information Security Policy" relate mainly to human processes and not infrastructure systems. Hence, we see that a significant portion of the policies defined in the PCI-DSS requirements are related to infrastructure security policies.

As a case study, we consider three policies taken from requirement 1, requirement 2, and requirement 6, and we focus on defining concrete monitoring rules that can be checked at runtime using an event-based system. We show our example policies in Table 3.3. For each policy, we list the information sources providing information for the validation of the policy.

First, we identify the events needed for the runtime validation of compliance. Such events are examples of possible information generated by the infrastructure, and they are not the only possible events that can be generated. Different monitoring system can generate a different set of events. Additionally, for simplifying policies, a few events represent "high-level" data inferred from lower level details.

The first policy requires "restricting inbound and outbound traffic to that

Table 3.3: Example of PCI-DSS policies. For each policy, we list the information sources that we assume available for its monitoring.

ID	Description	Sources
1.2.1	Restrict inbound and outbound traffic to that which is necessary for the cardholder data environment.	network, configurations
2.2.1	Implement only one primary function per server to prevent functions that require different security levels from co-existing on the same server.	SNMP, configurations
5.2	Ensure that all anti-virus mechanisms are current, actively running, and generating audit logs.	SNMP, vendor data

which is necessary for the cardholder data environment." The monitoring for compliance to such a policy requires obtaining information about network flows, about the machines that are part of the cardholder data environment, and about explicitly approved flows. Information about the network flow is generated by network monitoring systems, while the other pieces of information are generated by software management systems.

The second policy requires "implementing only one primary function per server to prevent functions that require different security levels from co-existing on the same server." Such a policy requires obtaining the list of services deployed on each server, and their security requirements. We obtain the list of software deployed on each server through a software management system, and we verify that such software is running using SNMP. The security levels are defined as attributes in the software management system.

The third policy requires "ensuring that all anti-virus mechanisms are current, actively running, and generating audit logs." Such a policy requires obtaining the list of software deployed on a machine, information about the latest version of software, and the presence of audit logs. We show a summary of the events in Table 3.4. While this is not the only set of events that can be generated to validate policies, the listed events are generally obtainable through standard systems.

We use the defined events to encode the first policy "restrict inbound and outbound traffic to that which is necessary for the cardholder data environment" using the following rules. We use the state-based representation.

Table 3.4: Monitoring events for the validation of PCI-DSS policies

Event and parameters	**Description**
`netflow`(IP_s, P_s, IP_d, P_d, $Proto$)	Detected a network flow from IP_s on port P_s to IP_d on port P_d with protocol $Proto$.
`cardEnv`(H)	A host H is part of the cardholder data environment.
`hasIP`(H, IP)	A host M has been assigned the network address IP.
`okFlow`(IP_s, P_s, IP_d, P_d, $Proto$)	The flow between IP_s on port P_s and IP_d on P_d for protocol $Proto$ is allowed.
`runs` (H, Sw, $Hash$)	Host H is running a software with name Sw and identified by $Hash$
`primarySw`(Sw, $Hash$, F, Sec)	The software Sw is a primary software for the function F (e.g., httpd for web server) and runs at security level Sec
`antiVirus`(Sw, $Hash$)	The software Sw, $Hash$ is classified as an anti-virus software
`latestAv`(Sw, $Hash$)	The software Sw with a binary $Hash$ is the latest version of the AV software available from the software provider.

$$\begin{aligned}&\texttt{netflow}(IP_s, P_s, IP_d, P_d, Proto),\texttt{hasIP}(H, IP_d),\\&\texttt{cardEnv}(H), \neg\texttt{okFlow}(IP_s, P_s, IP_d, P_d, Port)\\&\rightarrow\texttt{violations}(IP_d, P_d)\end{aligned} \tag{3.4}$$

$$\begin{aligned}&\texttt{netflow}(IP_s, P_s, IP_d, P_d, Proto),\texttt{hasIP}(H, IP_s),\\&\texttt{cardEnv}(H), \neg\texttt{okFlow}(IP_s, P_s, IP_d, P_d, Port)\\&\rightarrow\texttt{violations}(IP_d, P_d).\end{aligned}$$

The second policy, "implement only one primary function per server to prevent functions that require different security levels from co-existing on the same server," can be specified using alternative rules, depending on the semantic that we want to enforce. We use the following policy to enforce the application of a single primary function for each server, independently from the security level.

$$\begin{aligned}&\texttt{primarySw}(Sw_1, Hash_1, F_1, SecLev_1),\texttt{primarySw}(Sw_2, Hash_2, F_2, SecLev_2),\\&\texttt{runs}(H, Sw_1, Hash_1),\texttt{runs}(H, Sw_2, Hash_2),\\&\rightarrow\texttt{violation}(H).\end{aligned} \tag{3.5}$$

We can relax this policy so that it is violated only if the two services are of different "security levels" (e.g., a cardholder service and an email server). We use the following policy.

$$\begin{aligned}&\texttt{primarySw}(Sw_1, Hash_1, F_1, SecLev_1),\texttt{primarySw}(Sw_2, Hash_2, F_2, SecLev_2),\\&\texttt{runs}(H, Sw_1, Hash_1),\texttt{runs}(H, Sw_2, Hash_2),\\&\texttt{notEqual}(SecLev_1, SecLev_2) \rightarrow violation(H).\end{aligned} \tag{3.6}$$

The predicate `NotEqual` is true when the values of the two variables are different. The third policy, "ensure that all anti-virus mechanisms are current, actively running, and generating audit logs" is represented using the following conditions.

$$\begin{aligned}&\texttt{runs}(H, Sw, Hash),\texttt{antivirus}(Sw, Hash, F, SecLev),\\&\neg\texttt{latestAV}(Sw, Hash) \rightarrow\texttt{violation}(H).\end{aligned} \tag{3.7}$$

In general, we see how these policies require the correlation of only a limited amount of events. The first two policies require four events, while the third policy only three events. Our experience show that such a case is common for other infrastructure policies, even if our system can support more complex policies.

3.7 Policy Example: XDS and ATNA

The *Integrating the Healthcare Enterprise initiative* [57] provides a set of standards and protocols for interoperability between enterprises operating in the health care domain. The initiative is structured in a set of profiles, each providing interoperability in a specific portion of the health-care interaction. For example, a profile specifies protocols and workflows for sharing radiology data across enterprises, and another profile for cardiology. Our analysis focuses on two interconnected parts of such an initiative: the Cross-Enterprise Document Sharing (XDS) and the Audit Trail and Node Authentication (ATNA) integration profile. The XDS provides the architecture and the protocols for the exchange of information, while the ATNA profile provides protocols for the authentication of hosts and for creating audit logs. Such two initiatives provide a significant example of interactions between organizations and of

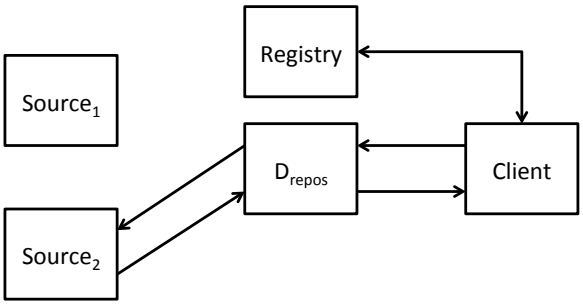

Figure 3.4: IHE Architecture for accessing health record stored in multiple locations.

the need for integrating audit logs generated by multiple sources.

The architecture for XDS is shown in Figure 3.4. When a client requests access to information for a patient, it looks up its ID in a common registry. The result of the query contains the patient's unique ID in the system, and the list of locations (i.e., *document sources*) containing information about the user. This information is then passed to a broker (i.e., the *document repository*) which does not contain any document, but acts as an intermediary for containing the original systems containing the patient data. The document repository acquires the information from the document sources and provides it to the client as response to the request.

We use an event-based representation and we provide examples of two policies.

The first policy specifies that a client needs to have a relation with a patient in order to access the patient ID (e.g., the ID requested needs to be a patient seeking healthcare services from the client). If the registry is in charge of monitoring for compliance to such a policy, information about the patient relation needs to be shared by the client.

We consider two entities: the client and the registry. We assume that the registry creates a local audit event indicating that a request for a patient id p_{id} is submitted by a client c. We indicate such an event as follows.

$$\begin{aligned}&(E,\texttt{type},\texttt{Patient Identity Feed}),\\&(E,\texttt{userId},c),(E,\texttt{patientId},p_{id})\end{aligned} \tag{3.8}$$

The client maintains audit logs indicating the patients with which it has a relation. For example, a relation between patient and client is inferred by the presence of a patient record in the client's database. We are interested

in the final audit event containing such information, and we represent it as follows.

$$\begin{array}{l}(E,\texttt{type},\texttt{Patient Relation}),\\(E,\texttt{userId},c),(E,\texttt{patientId},p_{id})\end{array} \tag{3.9}$$

We define the policy that requires the first event to occur only while the relation exists. Hence, we represent it as follows.

$$\begin{array}{l}(E_1,\texttt{type},\texttt{Patient Identity Feed}),\\(E_1,\texttt{userId},C),(E_1,\texttt{patientId},P_{id})\\(E_2,\texttt{type},\texttt{Patient Relation}),\\(E_2,\texttt{userId},C),(E_2,\texttt{patientId},P_{id})\\\neg E_2 \wedge [E_1 \texttt{DURING} E_2] \rightarrow \mathit{violation}(P_{id},C).\end{array} \tag{3.10}$$

The second example policy we consider related to the ATNA protocol for acquiring documents from the document sources. The main requirement of the ATNA profile is to allow for hosts to pre-fetch healthcare documents before the authorized users logs into the host. Such a pre-fetching is useful to reduce the waiting time for doctors while they are visiting patients, or in case of emergency or overload situations, where the network operation is limited or destroyed. Using such a protocol, documents are transferred in advanced and removed only they are no longer needed.

Using the ATNA protocols, secure nodes in different domains authenticate using host-authentication and exchange health data. The data is stored locally, and then made available to users within a domain according to access control policies. We consider a process for auditing a connection between servers and ensure that the interaction is compliant with the security policy. We use audit logs to detect unauthorized access of restricted information from one of the secure nodes.

We consider the following four audit events. First, an event `userAuth` specifying the successful authentication of a user in the local enterprise system. Second, an event `nodeAuth` to specify the successful authentication of a node. Third, an event `queryImage` to indicate that that an image has been requested. Forth, an event `retrieveImage` to indicate the request of retrieval of an image.

The policy can be violated in multiple ways. A violation is an access to an image from a user which has not received local authentication. Figure 3.5

$(E_1$,type,retrieveImage$), (E_1$,status,success$), (E_1$,user$, U)$,
$(E_1$,fromSecureNode$, N_1), (E_1$,image$, I)$

$(E_2$,type,retrieveImage$), (E_2$,status,success$)$
$(E_2$,node$, N_1), (E_2$,fromSecureNode$, N_2), (E_2$,image$, I)$

$(E_3$,type,nodeAuth$), (E_3$,status,Success$)$,
$(E_3$,node$, N_1), (E_3$,bySecureNode$, N_2)$

$(E_4$,type,userAuth$), (E_4$,status,Success$)$
$(E_4$,user$, U), (E_4$,bySecureNode$, N_1)$

$[E_4 \texttt{before} E_1] \wedge [E_3 \texttt{before} E_2] \wedge [E_2 \texttt{before} E_1]$
$\neg E_4 \rightarrow \texttt{violation}(N_1, N_2, I)$

Figure 3.5: Policy for validation ATNA profile compliance.

shows the encoding of such a policy in our event-based representation.

The rule specifies that there is a violation if a user U accesses to the image I acquired by an authorized node, but an event that authorized the user in the local domain is not found. Administrators can define further policies to cover different situations that would create violations.

The events used for the validation of the ATNA policy are generated by different organizations. While the ATNA profile currently requires all access to the logged in an audit log for analysis, our approach is able to coordinate the sharing of such events and automatically ensure that only event relevant to the violation of the policy are shared across the organization boundaries.

3.8 Policy Example: Aerospace Domain

A recent vision in commercial aviation is the e-Enabled aircraft that operates as an intelligent, mobile node in the Internet [58]. Safety, security, environmental, efficiency and economic benefits are provided to airlines, crew, ground personnel, passengers and business stakeholders. The performance of future eEnabled fleets, however, depends on the correct operation of network services provided by airport infrastructure [59]. As the complexity of airport infrastructure increases and changes become more frequent, manual control and operation become ineffective. Unintended or malicious changes in the configuration of a single system could introduce errors in the overall

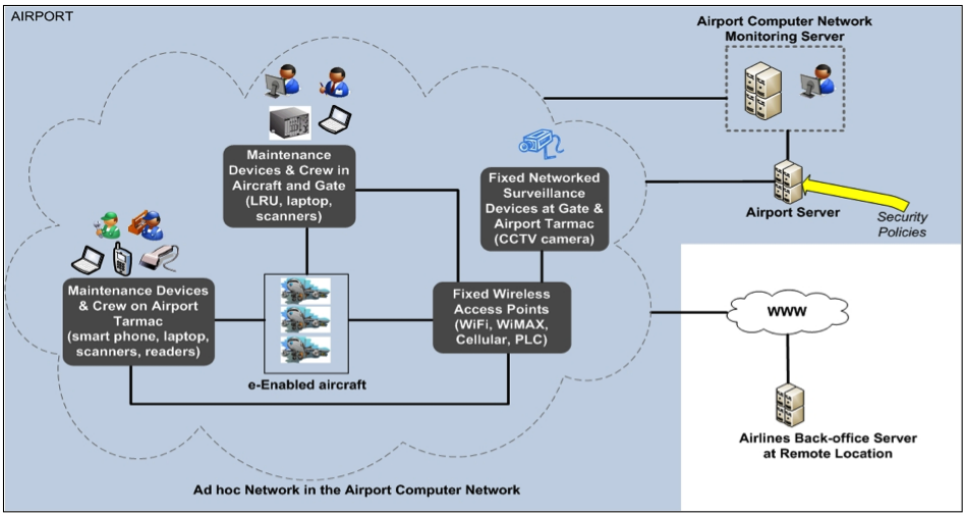

Figure 3.6: Architecture of the infrastructure of a future airport system

infrastructure configuration that might remain invisible and degrade or disrupt operations. Hence, for preserving performance gains of e-Enabling, such errors must be timely detected to prevent systems from operating in insecure or inefficient states.

A policy-based approach can provide such a detection. However the network infrastructure of airports supports several applications, from interconnecting devices at check-in desks and gates to supporting the communication between aircraft and airline systems. Such applications involve multiple organizations that closely interact. We focus on the interactions between e-Enabled fleets, maintenance personnel, and airport infrastructure services. Figure 3.6 provides an abstract view of the system architecture that we consider representative of the e-Enabled airline system architecture at airport. To support the maintenance of the e-Enabled fleet, four classes of devices interact in the airport infrastructure:

1. Aircraft Hardware Maintenance: Portable devices located on the tarmac that access aircraft systems to perform net-enabled operations (e.g., hardware inspection/repair, sensor readings). These typically belong to organizations that are leased or owned by airlines.

2. Connectivity: Fixed wireless/wired network access points (e.g., WiFi, WiMAX, cellular, and power line) that interconnect e-Enabled aircraft with off-board systems on the Internet. These typically are owned by third party service providers.

Table 3.5: Examples of policies specified to ensure safety of aircraft.

A.1. SAFETY RELATED POLICIES
(1) Approaching aircraft must establish a broadband wireless link when it enters weight-on-wheels condition. (2) Airplane accessing a ground-only airline application must be in weight-on-wheels condition. (3) Airplane must be parked at gate when accessing airline application Y. (4) RFID tags in cabin must not be read when airplane is not parked at gate.

3. Airline Maintenance Applications: Portable devices that are used in the cabin and gate to communicate directly with aircraft systems for airline maintenance applications (e.g., software update, cabin access). These are typically owned/leased by the airlines.

4. Tarmac Security Monitoring: Physical security services provided by CCTV camera, automatic door locks, and emergency devices. These are typically owned by the airport authority.

For supporting maintenance, we consider crew devices that (i) maintain the aircraft RFID-tagged hardware; (ii) manage software, and multimedia; and (iii) manage gates and control access to the aircraft cabins. Additionally, our system model considers other IT infrastructure that interacts with these devices. We include the interfaces with airline back-office servers that reside remotely and provide information, software and multimedia for the airline fleet. Security monitoring includes all physical security devices such as cameras or devices that enforce crew access control in restricted areas of / around the aircraft. Such devices communicate using the network infrastructure provided by the airport.

Policies defined to ensure the correct operation of the entire infrastructure can be organized in several classes. A few examples developed in collaboration with Boeing Research and Development are listed in Tables 3.5, Table 3.6, Table 3.7, and Table 3.8.

Monitoring for compliance to such policies requires interacting across several security domains. Airport systems are heterogenous, and several organizations interact to provide the necessary services (e.g., airport management, airlines, maintenance contractors). The information required for validating such policies comes from devices under the control of different entities and

Table 3.6: These policies are specified to prevent unauthorized access to aircraft.

A.2. ACCESS CONTROL POLICIES
(5) Maintenance device must be disconnected from Internet when accessing aircraft systems. (6) Maintenance crew must login with proper credentials (such as passwords) in order to be able to use maintenance laptop to perform assignments. (7) Users logged in maintenance laptop as maintenance crew cannot access files or perform security actions privileged to administrator only. (8) Maintenance crew and device credentials must be authorized and authenticated before interacting with airplane systems. (9) Maintenance crew and device must be located on airport tarmac when accessing external access point of aircraft. (10) Maintenance crew and device must be located inside cabin when accessing internal access point of aircraft.

Table 3.7: These policies are specified to minimize operational costs of aircraft while maintaining quality of network connection.

A.3. BUSINESS RELATED
(11) Aircraft must automatically choose the most cost-effective wireless data link available that satisfies the minimum bandwidth requirement. (12) When current bandwidth drops below the minimum bandwidth threshold, aircraft must automatically search for the next cheapest available wireless data link. (13) Maintenance devices should use the cheapest available wireless network access point to connect to Internet. (14) RFID tag must not respond to a query issued less than a threshold distant to protect airlines proprietary data. (15) While taxing, for a period of time, an airplane is allowed to be associated with two access points for smooth handover.

Table 3.8: These policies are specified to enable efficient resource allocation at the airport.

A.4. AIRPORT OPERATION RELATED
(16) Technology based requirement. Examples: Airplane using WiFi technology must not use Terminal T1. Airplane using Cellular technology must use Gates G1-G5 at Terminal T2. Airplane using WiMAX must use runway R1 for takeoff. (17) Airport layout based requirement. Examples: Airplane using Gates G10 to G20 must use WiMAX technology. Airplane using runway R2 must not use WiMAX technology. Crew at tarmac below Gates 30-35 must not use Cellular. (18) Time-based (network/airport demand based) requirement. Examples: Airplane using the airport computer network at terminal T3 between time t1 to time t2 must use WiMAX. Airplane using the airport computer network at Gates G40-G45 must not use WiFi.

needs to be integrated while respecting the privacy of the each organization.

In the aerospace domain, the increasing reliance of modern aircrafts on the airport infrastructure requires such systems to operate securely, safely, and efficiently. Infrastructure policies can be used to monitor that the infrastructure is operating in a good state. For example, the new e-Enabled fleets require services that need to be provided by the airport infrastructure, such as Internet connectivity, aircraft software update services, and access to maintenance services.

We analyze the opportunity of creating policies in such a context. An example policy can mandate that approaching airplane must establish wireless link after they land (i.e., weight-on-wheels condition) so that software updates can be downloaded, as following.

$$\begin{aligned}&(A,\texttt{landed},R),(R,\texttt{part\_of},AIR),(N,\texttt{part\_of}AIR),\\&\neg(A,\texttt{connected\_to},N)\rightarrow(policy_1,\texttt{violation},A),\end{aligned}\tag{3.11}$$

where the statement $(A, \texttt{landed}, R)$ indicates that the aircraft A landed on runway R, the statement $(K, \texttt{part\_of}, J)$ indicates that an entity K is part of the system J, and the statement $(A, \texttt{connected\_to}, N)$ indicates that the system A is connected to a network N. The consequence of the rule, $(policy_1, \texttt{violation}, A)$ indicates that a violation of $policy_1$ has been detected and it involves the aircraft A.

Additionally, we can monitor that airplane must be parked at gate when accessing certain airline applications, as following:

$$\begin{aligned}&(A, \texttt{logged\_on}, P), (P, \texttt{provided\_by}, M),\\&(M, \texttt{part\_of}, airline), (P, \texttt{use}, \texttt{gate\_restricted}),\\&\neg(A, \texttt{located\_at}, L), (L, \texttt{location\_type}, \texttt{gate})\\&\rightarrow (policy_2, \texttt{violation}, A),\end{aligned} \tag{3.12}$$

where the statement $(A, \texttt{logged\_on}, P)$ indicates that the aircraft A is accessing application P, $(P, \texttt{provided\_by}, M)$ indicates that the host M is running the application P, $(A, \texttt{locate\_at}, L)$ states that the aircraft A is located at location L. The last two statements, $(L, \texttt{location\_type}, \texttt{gate})$ and $(P, \texttt{use\_restricted}, \texttt{gate})$ state that L is a airport gate, and that P can be used only by an aircraft located at the gate.

All these example policies require integrating information across multiple devices for assessing compliance. The first policy requires integrating information from the aircrafts or the control tower with information generated by network monitoring devices, while the second policy requires integrating again information from the aircraft and information generated by airline applications. However, none of these policies requires access to the entire state of the system. Our architecture enabled an organization to acquire information from the devices it manages, and exchange a limited set of events with other organizations to ensure overall compliance.

CHAPTER 4

LOCALITY-BASED APPROACHES

For ensuring the completeness and the correctness of the monitoring process, an organization needs to ensure that the need-to-know events are among the events shared with the other organization. However, identifying such a minimal set is challenging, as it depends on the effect of each event on the compliance state of the other organization. We have a tradeoff in sharing: the more an organization shares information about its events, the better the other organization can reduce the events to share by better identifying the need-to-know events. Generally, more information than the minimum need-to-know needs to be exchanged across organizations.

In this thesis, we present several techniques to guide the interaction between organizations and limit the information exchanged between organizations. Our techniques are based on two fundamental concepts. The first concept is *locality* of information. We use the description of what information is completely observable within a single system to identify which violations are identified locally, and to summarize locally information to reduce the information shared. The second concept is *resource*. A resource is any entity within a computer infrastructure system, such as computer systems, virtual instances, users, or programs. Violations of policies can be seen as conditions regarding a limited set of interacting resources that operate outside the conditions specified in the policy. Our technique identifies such subset of resources and limits the exchange of information only across those.

This chapter focuses on the first concept: locality. We identify the policy violations that need to be detected within each organization, and we use meta-data about the information collected by each monitoring system to identify the information to collect from other external entities. Based on such knowledge, we introduce two protocols for obtaining the remaining information from other organizations and ensure that policy monitoring detects all violations.

Additionally, we show that such a locality-based approach can be extended also within each organization to reduce significantly the information collected by the monitoring systems. We introduce an algorithm that uses metadata about the information collected by each device to distribute information processing across the network.

4.1 Overview

The locality-based approaches are based on exploiting knowledge about the types of information collected from the monitoring system. If the events that validate a portion of the policy are all observable locally and cannot be generated by any other organization, there is no need of exchanging information with other organizations to ensure the completeness and the correctness of such a portion of the policy.

This section provides a rule-rewrite techniques for separating rules into a local portions that can be completely validated within an organization. Information exchange is necessary only for the parts of the policy that are not local. We focus on the case of two organizations and we introduce two protocols that guide the interaction between organizations for the validation of policies. In the first protocol, an organization acquires from the other organization all the information required for the validation of a policy without revealing information about its state. In the second protocol, each organization shares some information about its own state to validate the overall compliance.

The process of detecting violations is as follows. An organization A interested in detecting a set of violations V_A analyses its policies and its past events to create a set of persistent queries P over the stream of events of the other organization. Events in B matching queries are continuously sent back to A. Collectively, these events form a set of events E' that is guaranteed to contain E.

4.2 Policy Violations of Local Resources

In the context of infrastructure monitoring, monitoring systems such as network management systems, IDS, or anti-viruses generate events that carry information about some *resources* in the system. The monitoring system of an organization A receives a portion of the events of the entire system. In particular, it receives events that carry information about the resources under the control of organization A.

An organization is generally interested in detecting policy violations that relate to its own resources. For example, in the case of a simple policy requiring a host-based IDS to be running on each machine, an organization might be interested in receiving violations only for the machines that it manages. We associate each resource in the system to a domain. We assume that each resource has a unique name (i.e., a URI). We define a set U containing all resources in the overall system, and its subsets D_A and D_B representing the sets of resources owned or managed by each organization.

A policy defined over the entire set of resources U can be rewritten into two (or more) local policies, each of them restricting the values of certain variables in the domains D_A or D_B.

An organization defines the violations of interest by specifying—in the policy itself—the domains over which variables are quantified. For example, a policy can specify that a violation occurs if a device running a critical service connects to a server that is vulnerable. A monitoring system can collect events from multiple information sources. In our example, the state of the system is represented as the following set of facts. The software running on a device (e.g, `runs` $(host_1, pid_{1,111}, apache2.4)$), the network connections that are open by the given programs (e.g., `connects` $(pid111, host_2, 3111)$), and the service running on specific ports (e.g., `binds` $(host_2, pid_{2,321}, 3111)$) are collected using SNMP monitoring. The presence of a vulnerability in the software (e.g., `vulnerable`$(mysql5.0)$) is collected from the National Vulnerability Database (NVD) [60]. Finally, the list of critical software (e.g., `critical`$(apache)$) is provided by the user using an online system. The overall rule is represented as follows.

$$\begin{aligned}&\texttt{violation} \leftarrow \texttt{runs}(X, P_1, S_1), \texttt{critical}(S_1),\\&\texttt{connects}(X, P_1, Y, PORT), \texttt{binds}(Y, P_2, PORT),\\&\texttt{runs}(Y, P_2, S_2), \texttt{vulnerable}(S_2)\end{aligned} \tag{4.1}$$

Such a rule specifies that a violation exists if there is a machine X running a critical software S_1 that is connected to a vulnerable program P_2 running on another machine Y. We can add domains to such a policy to specify that organization A is interested in receiving all violations about its own machines as follows:

$$\begin{aligned}&\forall X \in D_A, \forall P_X, S_X, Y, P_Y, S_Y, PORT \in U\\&\texttt{violation}(X, S_X) \leftarrow \texttt{runs}(X, P_X, S_X), \texttt{critical}(S_X),\\&\texttt{connects}(X, P_X, Y, PORT), \texttt{binds}(Y, P_Y, PORT),\\&\texttt{runs}(Y, P_Y, S_Y), \texttt{vulnerable}(S_Y),\end{aligned} \tag{4.2}$$

where X and Y are hosts; P_X and P_Y are PIDs of processes; S_X and S_Y are identifiers of software packages; and $PORT$ is a network port number. We identify the interest in violations involving resources of organization A by restricting the value of the variable X to D_A. However, we cannot restrict the domain of other variables. For example, restricting the domain of Y to D_A would make the policy identify only violations that involve connections between two hosts within A. By restricting only the domain of the variables in the head of the rule (i.e., in the violation statement), we ensure that we find all violations in K^+ that relate to resources in D_A.

4.3 Completeness of Local Information

The monitoring system of an organization focuses on acquiring events from a specific set of resources: the resources under the control of the organization. For such resources, we might be able to acquire *complete* information. For example, we can consider a simple event in a monitoring system that indicates if a network card is overloaded with data (i.e., it is losing a significant amount of packets becomes of intense traffic). A monitoring system can indicate such as situation by generating events $\texttt{overloaded}(H)$. The set of events $\texttt{overloaded}(H)$, with $H \in D_i$, is complete if it is monitoring the state of all hosts in D_i. If a monitoring system collects all information about resources in its domain, D_i would be equal to D_A. In such a case, if a policy needs

to match events $\texttt{overloaded}(H)$ with $H \in D_A$, all the relevant events can be found on the local knowledge base K_A. Hence, for complete statements, the closed-world assumption of Datalog holds in the local KB and reasoning based on them is correct and complete.

We model the completeness of knowledge acquired by the monitoring system with a *completeness KB* (CKB). A CKB describes patterns of events about which the local monitoring system ensures that we have complete knowledge. A CKB depends on the structure of the local monitoring system and it is composed of two types of statements: simple completeness statements and conditional completeness statements.

A *simple completeness statement* is a pattern defining events for which we have complete local knowledge. If a policy is looking for a certain pattern of events and such a pattern can be described by a simple completeness statement, then all events matching it can be found in the local KB. A simple completeness statement is expressed by a statement and by the domains of its variables. We indicate it with the syntax $\forall V_i \in D : st(V_1, \ldots, V_n)$. A statement $b_i(U_1, \ldots, U_n)$ matches a simple completeness statement $st(V_1, \ldots, V_n)$ when $b_i = st$ and for all $i, 1 \leq i \leq n$, $domain(U_i) \subseteq domain(V_i)$. For example, the simple completeness statement for the `overloaded` event above is $\forall H \in D_A : \texttt{overloaded}(H)$.

A *conditional completeness statement* provides a restricted notion of completeness of an event pattern. For example, we can define a set of events $\texttt{critical}(P)$ indicating that a particular software P is currently critical to the organization operations. By using simple completeness statements, we would be able to express that, given any program P, a monitoring system has generated events indicating if such program is critical to the system as a whole. However, most monitoring systems would be able to decide only which programs are critical to the local organization, and might not be aware of the critical programs for the other organization. We represent such restriction of knowledge by using *conditional completeness statements*. In our example, we express that a monitoring system generates such `critical` events only for the programs that are currently running on its devices by specifying the following: $\forall X \in D_A, \forall P \in D : \texttt{critical}(P) \leftarrow \texttt{runs}(X, P)$. Such statement indicates that, given a policy containing an event pattern $\texttt{critical}(P)$, we can consider the pattern local only if the values of P are restricted to the programs running on the machines in the domain D_A. In general, we represent

$\forall X \in D_A : \texttt{overloaded}(X)$
$\forall X \in D_A, P, PID \in U : \texttt{runs}(X, PID, P)$
$\forall X \in D_A, PID, P \in U : \texttt{critical}(P) \leftarrow \texttt{runs}(X, PID, P)$

Figure 4.1: Example of completeness KB for organization A containing simple and conditional completeness statements

conditional completeness statements as $\forall V_i \in D : st \leftarrow c_1, \ldots, c_m$.

Given a policy, our information sharing strategies start by performing a *completeness analysis* to identify the information provided by the local monitoring system. The completeness analysis takes a policy $v \leftarrow p_1, \ldots, p_n$, a completeness knowledge base CKB, and determines which statements p_i can be found completely on the local knowledge base K_i. We call such statements *local-complete*. We indicate the set of local-complete statements with S_L, and use S_R to indicate the others. A non-empty S_R indicates that events affecting the result of the policy compliance process might be stored in K_B.

For example, the following policy can be answered completely in the knowledge base subject to the CKB in Figure 4.1.

$$\begin{aligned} &\forall M \in D_1, \forall S \in D \\ &\texttt{overloaded}(M), \texttt{runs}(M, PID, S), \texttt{critical}(S) \\ &\rightarrow \texttt{violation}(M, S). \end{aligned} \tag{4.3}$$

The statements `overloaded` and `runs` are complete because of the simple completeness conditions in the CKB. The statement `critical` is complete because of the conditional completeness statement $\texttt{critical}(\ldots) \leftarrow \texttt{runs}(\ldots)$. In this case, $S_R = \emptyset$ and $S_L = \{\texttt{overloaded}(M), \texttt{runs}(M, PID, S), \texttt{critical}(S)\}$.

If the set S_R is not empty we need to acquire all relevant events from K_B. We introduce two strategies for defining the set of persistent queries that provide a complete and correct detection of all policy violations. The first strategy, called *asymmetric pull strategy*, creates a set of queries P from the set S_R. Such a set of persistent queries is independent from the events contained in K_A and does not provide B any information about events. The second strategy, a *symmetric push-pull strategy* determines the queries in P using both S_R and the current state K_A of the system. Added or removed events in K_A can add or remove queries. Using the symmetric strategy, A can reduce the amount of events requested from B at the cost of revealing

some of its internal state.

4.4 Asymmetric Pull Protocol

The first protocol uses the set S_R to acquire from K_B all events relevant to the policy. If the statements in $S_R = \{p_1, \ldots, p_n\}$ are simple events, creating a set of queries $P = \{(p_1), \ldots, (p_n)\}$ is sufficient for ensuring completeness and correctness: all events matching any of the statements relevant for the policy that cannot be found completely on K_A are sent back to A. Such a process guarantees that all events potentially relevant to the process are known by A.

When the statements in S_R are complex events, we need to analyze recursively the policy to identify all simple and complex events that can contribute to it. In our proofs, we take advantage of the connection between complex event processing and relational algebra formalized by Bry et al. [61]. Given a knowledge base K, we use the symbol $\sigma_{p_i}(K)$ to indicate a knowledge base obtained by selecting only statements in K for which there exists a substitution of variables that unify with the given statement p_i. Given a set of statements B, we define the *extended set* B' by adding to the set B the statements d_j contained in the rules $h \leftarrow d_1, \ldots, d_m$ where a $p_i \in B$ unifies with h. We continue until we analyzed all statements in B'.

Lemma 1. *We take a violation v, a rule $v \leftarrow p_1, \ldots, p_n$, a knowledge base K containing ground statements and the sets of rules R. We consider a set $B = p_1, \ldots, p_n$ and its extended set B'. We have that $K \vdash v \Leftrightarrow \cup_{b_i \in B'} \sigma_{b_i}(K) \vdash v$.*

Proof Sketch. Proving ($\Leftarrow$) is simple: $\bigcup_{b_i} \sigma_{b_i}(K)$ is a subset of K, so if it can prove v then K can prove v. The other direction can be proven by contradiction: assume that there is an event not in B' that is necessary in the proof of v. This event is either part of the policy or generated to contribute eventually to the policy. However, events of this type are included in B' by definition. □

Theorem 2. *Given two sets K_A, K_B, a rule $V = \{v \leftarrow p_1, \ldots p_n\}$ with a body containing $S_L = \{p_1, \ldots, p_{r-1}\}$ statements and $S_R = \{p_r, \ldots, p_n\}$ statements. Let S'_R be the extension of S_R on the rules R.*

For every v we have that $(K_A \cup (\bigcup_{b_i \in S'_R} \sigma_{b_i}(K_B)) \cup R) \vdash v \Leftrightarrow K^+ \vdash v$.

Proof Sketch. For $P = S_L \cup S'_R$, the lemma implies $(\bigcup_{p_i \in P}(\sigma_{p_i}(K_A \cup K_B)) \cup R) \vdash v \Leftrightarrow K^+ \vdash v$. By definition of completeness, for each $p_i \in S_L$, we have that $\sigma_{p_i}(K_A \cup K_B) = \sigma_{p_i}(K_A)$. Hence, we can rewrite the first part of the expression as $\bigcup_{p_i \in P} \sigma_{p_i}(K_A) \bigcup_{p_i \in S'_R} \sigma_{p_i}(K_B) \wedge R$. Because K_A is a superset of $\bigcup_{p_i \in P} K_A$, we can rewrite the entire expression and obtain $(K_A \cup (\bigcup_{p_i \in S'_R}(K_B)) \cup R) \vdash v \Leftrightarrow K^+ \vdash v$ □

4.5 Symmetric Push-Pull Protocol

The second protocol uses the set S_R and the events in K_A to create a set of persistent queries P selecting a narrower set of events E'. This narrower selection is obtained by providing to B some limited information about K_A so that only events that are potentially relevant to a violation are delivered to A.

Intuitively, the strategy is based on selecting the events matching on K_A the local part of the policy to determine a set of specific "missing events" that, if present in K_B, would create a violation. In particular, we consider the set $S_L = \{p_1, \ldots, p_{r-1}\}$ as a query $L = (p_1 \wedge \ldots \wedge p_{r-1})$. For each set of events matching the query, we substitute the values of the variables (substitution γ_i) in the non-local statements S_R. After this process, we call statements in $\gamma_i(S_R)$ that now have at least a ground parameter *boundary statements*. These statements are submitted as persistent queries to organization B. Events matching these queries are returned to A. A then adds the new values for the variables to each substitution γ_i and repeats the process until all statements are considered.

1. We take a policy $v \leftarrow p_1, \ldots, p_n$. We assume that all p_i are simple events. We will see below how to generalize it to complex events.

2. Starting from S_R and S_L of the policy, we compute the set S_B of statements in S_R that share at least one variable with statements in S_L. S_B is the *boundary set* which represents the remote information about which the local statements have partial knowledge. If we indicate with X_L the variables used in S_L and with X_R the variables used in S_R, we can determine the set of shared variables $X_B = X_L \cap X_R$. If $X_B = \emptyset$ but $S_R \neq \emptyset$, no variables are shared and we revert to the pull algorithm.

3. We construct a local query on K_A by taking the conjunction of the statements in S_L and projecting the result on the variables X_B. We substitute the variables in $S_B = \{p_1, \ldots, p_b\}$ with the substitution γ_i and we create a set of queries $P = \{(\gamma_i(p_1)), \ldots, (\gamma_i(p_b))\}$. The queries P are submitted to organization B.

4. When new events are received from B, we add the results in the knowledge base and create a new CKB' where we add a conditional completeness condition $\bigwedge_{l_i \in S_L} l_i \rightarrow p_i$ for each $p_i \in S_B$ where $p_{l,i} \in S_L$. We repeat the algorithm with the new CKB' until $S_R = \emptyset$

If an event p_i is a complex event, we add an additional step to the process. We consider the rule heads that unify with p_i, and we apply the algorithm recursively to the respective rules.

The correctness and completeness of this process can be shown with a few considerations. First, if we cannot find a substitution γ_i satisfying S_L on K_A, we have that $K^+ \nvdash v$. This comes from the local completeness of S_L: events in S_L can be found only in K_A or they do not exist in K. If we have a substitution γ_i, we can consider it a partial match of a rule. If we can find events matching the statements $p_j \in S_R$ in K_B that are compatible with the substitution γ_i, then we have found a set of events matching the condition of the rule. By submitting the persistent queries we obtain such a result: either the event occurs and it is delivered to A and added to a K'_A, or it does not occur. In either case, K'_A is now complete in respect to the new events. As we send queries for all γ_i, we can add a new completeness statement $\bigwedge_{l_i \in S_L} l_i \rightarrow p_j$.

Lemma 2. *Given a policy $v \leftarrow p_1, \ldots, p_n$, two sets $S_L = \{p_1, \ldots, p_{r-1}\}$ and $S_R = \{p_r, \ldots, p_n\}$, a set $S_B = \{p_r, \ldots, p_b\}$, a completeness KB CKB, and a set of substitution γ_i obtained by performing the query $p_1 \wedge \ldots \wedge p_{r-1}$ on K_A. We have that $K'_A = K_A \bigcup_{p_j \in S_B} \bigcup_i \sigma_{\gamma_i(p_j)}(K_B)$ is complete with respect with $CKB' = CKB \cup (\bigwedge_{l_i \in S_L} l_i \rightarrow p_j)$.*

Proof Sketch. For p_j to be conditionally complete in K'_A we need to ensure that for all events matching p_j for which there are a set of connected events $\bigwedge_{l_i \in S_L} l_i$, we have that $K'_A \vdash p_j \Leftrightarrow K \vdash p_j$. First, we show that if $K \vdash p_j \Rightarrow K'_A \vdash p_j$. We have that for all γ_i such that $\gamma_i(S_L) \subset K_A$, we submit a query to K_B and we obtain $\gamma_i(p_j)$. In this way we obtain all p_j for which the

condition $\bigwedge_{l_i \in S_L} l_i$ are true. Similarly, $K'_A \vdash p_j \Rightarrow K \vdash p_j$ because the set of queries can also identify the lack of the event. □

Theorem 3. *Given a policy $v \leftarrow p_1, \ldots, p_n$, two sets $S_L = \{p_1, \ldots, p_{r-1}\}$ and $S_R = \{p_r, \ldots, p_n\}$, and a completeness KB CKB, the push-pull protocol is complete, correct, and terminates.*

Proof Sketch. The core of the proof proceeds by induction. Because of space restrictions, we provide only a sketch. Each step of the protocol creates a set S_B^i and acquires the set of events $\bigcup_{p_j \in S_B^i} \bigcup_k \sigma_{\gamma_k(p_j)}(K_B)$ from K_B. From Lemma 2, the new knowledge base K_A^i is complete for $CKB^i = CKB^{i-1} \cup (\bigwedge_{l_k \in S_L} l_k \rightarrow p_j)$. Because of the conditional completeness, we have that $S_L^i = S_L^{i-1} \cup S_B^{i-1}$ (computed with the new CKB^i). We use the new S_L^i to compute a new S_R^i and S_B^i. By induction, S_L^i will increase in size by some nonzero amount for each i since S_B^i contains disjoint events by definition. Similarly, S_R^i will decrease in size. If $S_B^i \neq \emptyset$ during the protocol, eventually $S_R^i = \emptyset$ and the validation of compliance is performed locally on K_A^m, which is complete and correct by definition. If at any point $S_B^i = \emptyset$, the K_A^i is obtained by running the pull algorithm, which is known to be complete from Theorem 1. Since the universe of events is finite, S_L^i is finite and the protocol will eventually terminate. □

4.6 Device-based Rewrites

The pull and push-pull protocols can be used across organizations to collect the information required for identifying violations. The remaining part of this chapter introduces Odessa, a monitoring system that extends the use of locality and local processing across a large set of devices. Information about violations is processed locally by each device collecting data. Such a distributed processing can significantly reduce the information that needs to be stored in the centralized monitoring systems within each organization. This locality-based rewrite is used on each device generating information to identify portions of the policy that can be found locally. When an event source is observing a complete set of events for a subset of the policy, then processing can be done locally to reduce the load on the monitoring server, and to not store unnecessary information in a central location.

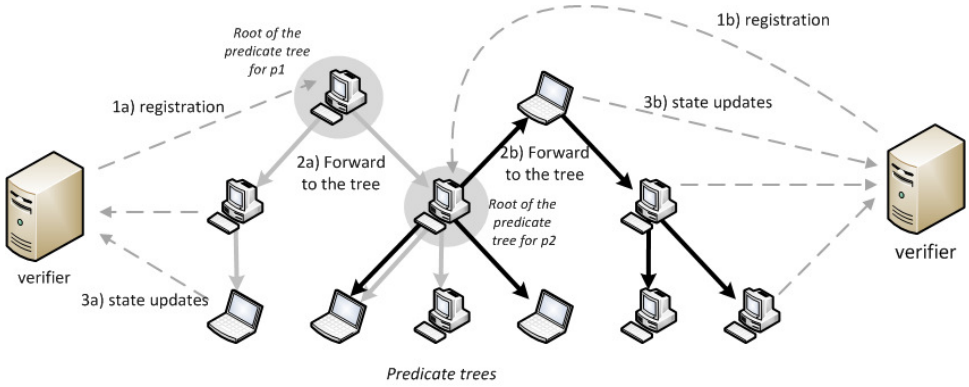

Figure 4.2: Architecture of Odessa. End-hosts are organized in predicate groups. Verifiers register to the root of the group.

The objective of Odessa is to check if the state of the infrastructure as a whole is compliant to all policies defined by the organization and represented as Datalog rules. The architecture of the system was designed to distribute the evaluation of these rules in a scalable manner. To achieve this, Odessa uses a distributed evaluation protocol which includes a set of *monitoring agents* that monitor the configuration of hosts and validate local portions of the organizations' policies and a set of distributed *verifiers* that validate global rules. The scalable coordination between monitoring agents and verifiers is performed using distributed index structures called *predicate groups*.

Agents represent the state of hosts using Datalog statements and share statements relevant to global policies with verifiers. Distributed verifiers integrate information across multiple hosts and perform the validation of policies with configuration information gathered from multiple machines. For a subset of policies critical to the security of the infrastructure, we require configuration information to be acquired independently from multiple agents and we replicate the policy validation on several verifiers which use Byzantine fault tolerance [62] to reach an agreement. By virtue of using FreePastry [63], our system inherits secured SSL communications and heartbeat-based liveness measurement. The architecture of Odessa is depicted in Figure 4.2.

4.6.1 Monitoring Agents

Monitoring agents run in a secure environment on each host. We developed agents running in the `Dom0` VM of a Xen system [64]. Virtual machines running on the same physical hosts are monitored using VM introspection [65]. Traditional hosts are monitored remotely by one of the agents using standard protocols. TPM can be used with SSL to provide a root of trust and validate the authenticity of the agent. The set of policy violations that our system detects depends on the type of information that monitoring agents can acquire from the systems.

With the increasing use of virtualization in modern systems, running a monitoring agent in a separated virtual machine is becoming easier. When virtualization is used internally within an organization to consolidate different servers, the deployment of Odessa only requires installing the monitoring agent in a separated monitoring VM (i.e., Dom0) co-located with the monitored hosts. The deployment of monitoring agents in environments where monitoring VMs are not controlled directly by the organization can be performed through the use of solutions that have been proposed in the research literature. One solution is to use nested layers of virtualization to create a virtualized monitoring VM that has control of the other VMs of the same user [66]; another solution is to have the cloud provider run monitoring agents and expose interfaces that can be used by the cloud customer to acquire monitoring data [67].

Monitoring agents maintain a list of policies defined by the organization. Such a list is periodically updated through a software update process orthogonal to the monitoring process.

4.6.2 Verifiers

The verifiers are the machines under the control of the security administrators that have the task of detecting violations of policies. Verifiers are hosts that securely collect information shared by monitoring agents in order to validate rules. Each verifier manages a subset of the rules of the organization. The set of rules is partitioned across verifiers to minimize the overlap between the information required by each verifier. Critical rules are analyzed redundantly by multiple verifiers. Verifiers interact with predicate groups to

receive information about the system by "subscribing" to events relevant to its rules.

4.6.3 Predicate Group

To link monitoring agents and verifiers, we use predicate groups. Each predicate group connects monitoring agents that share a specific type of configuration statements and, hence, participate in the evaluation of the same rules. These groups distribute the processes of disseminating information about new verifiers, integrate new hosts, and monitor their liveness.

The first task of a predicate group is to permit a connection between the verifiers and the agents holding the information needed for the verification of compliance. A verifier monitors compliance to a subset of the rules in the system, so each verifier is interested in receiving only a portion of the state change events generated by monitoring agents. We identify such a subset by considering the types of facts contained in each rule: if an event of type p is used in a rule, then the verifier is interested in receiving all state event changes related to p.

Example 1. *We can consider a simple rule which specifies that critical servers should not run applications with known vulnerabilities without an exception. By acquiring information about the running program on each machine, annotations about the importance of each server, and information about vulnerabilities, we represent this rule by specifying that we have a violation if a critical server provides a vulnerable service as following:* $(S, istypeserver, true)$, $(A, istypeservice, true)$, $(S, provides, A)$, $(S, criticality, high)$, $(A, hasvuln, V)$, $\neg(S, hasexception, true) \rightarrow (r_1, violation, V)$.

Example 2. *If we consider the rule in Example 1, a verifier can check compliance by receiving the events of type istypeserver, istypeservice, provides, criticality, hasvuln, and hasexception (next section will show how these events can be reduced by performing local processing).*

We organize predicate groups around predicates by associating a predicate group T_p to each predicate p. Membership in the group is distributed to several hosts by organizing agents into trees: each host maintains knowledge about a few other agents and monitors their liveness. The processes of

constructing and maintaining these trees are inspired by the Scribe dissemination trees [68]. Communications are built on a Pastry Distributed Hash Table (DHT) [69] system, and the agent assigned to the DHT key $H(p)$ is the root of the tree for predicate p. Such assignment is dynamic and depends on the availability of the agents. If the agent on which the key $H(p)$ is currently assigned becomes unavailable, the underlying DHT infrastructure automatically remaps the key to another agent in the system. Such new agent starts receiving the messages directed to $H(p)$.

Each monitoring agent joins predicate groups related to the statements contained in the configuration of the hosts they monitor. Using the simple example from the previous section, we have predicate groups for the types of predicates *istypeserver*, *istypeservice*, *provides*, *criticality*, *hasvuln*, and *hasexception* (in a real configuration, the number of predicates is larger). The agent is part of multiple predicate groups, one for each predicate p contained in the configuration, and maintains local information about each of them. This information includes the list of verifiers interested in the given predicate and the addresses of a few other agents in the same predicate group. On an agent a, we represent such information as $T_p^a = (V_p^a, U_p^a, C_p^a)$. V_p^a is a list of verifiers interested in receiving changes in the agent's state that relate to the given predicate; U_p^a is the address of an agent that is considered the *parent* of the a is the predicate group tree structure; and C_p^a is a list of agents within the same predicate group that are considered *children* of current agent.

When a new agent joins the system (e.g., a new physical host is added to a virtualized network), the agent needs to be included in the predicate trees. As in other peer-to-peer systems, we maintain a partial list of other long-lived agents in the system as a configuration of the agent. Once at least one live agent is found, the DHT reconfigures itself and obtains the addresses of other agents, if needed.

Predicate-group join protocol: For each predicate p in the configuration, the new monitoring agent joins a predicate group T_p by submitting a *subscribe request* message directed to $H(p)$ (i.e., the root of the tree for T_p). As in the Scribe protocol, such request is routed through the DHT toward the agent assigned to the key $H(p)$ and it is forwarded until it encounters an agent a' which is part of the same predicate group (in the worst case, a' is the tree root $H(p)$). More details can be found in Castro et al. [68]. Upon

reception, a' responds by sending its list of verifiers $V_p^{a'}$ and by adding the agent a to its list of agents $C_p^{a'}$. Upon reception of the response message, a sets its verifier list V_p^a to $V_p^{a'}$ and sets its parent node U_p^a to a'. Now a knows the verifiers interested in receiving information about its state. Hence, a sends its state to them and can keep them up-to-date about relevant state changes.

The membership in predicate groups is updated upon changes in the types of predicates monitored by an agent. When a new predicate p' is created in the configuration of one of the monitored hosts (e.g., because of a state change or because of the addition of a new monitoring capability), the monitoring agent joins the predicate group related to p' and receives the list of verifiers interested in such an information.

Verifier dissemination protocol: Verifiers interested in receiving information about a predicate p submit requests to the root node $H(p)$. The request contains the query and the address of the verifier so that responses can be routed directly to the destination. Once the request is received by $H(p)$, the agent disseminates the request to all monitoring agents through the tree structure of the predicate group: each agent a receiving the request resends it to the agents in its C_p^a. Once a monitoring agent receives the request, they add the new verifiers to their verifier list V_p^a. a sends its system state to the new verifier and continues to update the verifier in case of state changes. In this way, new verifiers can be easily added to the system: when a new verifier joins the system, it only needs to submit requests to the root of the predicate groups of interest.

Liveness monitoring: Parents and children hosts in a predicate group provide a distributed mechanism for checking the liveness of a monitoring agent. Maintaining correct information about liveness is critical for the verification of policies: if a host becomes inactive, its state should be removed from the system. For example, a policy could specify that the traffic of a subnet should be verified by an IDS. If the IDS fails, the policy would be violated. Detecting such a violation in a centralized system can be performed using periodic keep-alive. However, when the scale of the infrastructure becomes large, the frequency of keep-alive messages needs to be low to not overload the centralized server with network traffic. A low keep-alive frequency necessarily increases the time required for detecting failures and, hence, policy violations. By distributing the task of checking for liveness to a subset of

the hosts in a predicate group, the load on the centralized servers is reduced while the frequency of keep-alive checks can remain high. Parents and children in the predicate-group dissemination tree have knowledge of the verifiers interested in the state of the host and can notify them in case of failures.

The failure detection process is distributed through the exchange of keep-alive messages between parent agents and child agents at each level of the predicate group. When a failure is detected, an *agent-failure* message is sent to all verifiers in V_p^a. The verifiers check that the detected failure is not a false positive by attempting a communication with the agent. If such a communication is not possible, they remove the agent's state from the knowledge base.

In addition to the notification of failures to verifiers, the liveness monitoring is used to maintain the structure of the predicate group. When a child agent detects a failure of its parent, it executes the join protocol again to identify a new parent agent. Additionally, agents near the predicate group root detect if failures or additions of new agents triggered a reassignment of the root agent role. In such cases, the old state of the predicate group is preserved by having the child agent send it to the new root agent.

4.6.4 Rule Decomposition and Validation

To be scalable, our policy validation process detects policy violations through the coordination of monitoring agents and monitoring servers. We use our rule decomposition algorithm (RDA) to transform the organization's rules into an equivalent set of rules which takes advantage of information locality. This process allows Odessa to push a partial execution of the rules to each monitoring agent and hence, reduces configuration information transferred between machines.

The intuition behind the algorithm is to use information about the source of configuration statements (i.e., which agents can generate a particular configuration statement) for limiting the places where possible configurations that can trigger the body of a rule can be found. For example, if we are checking for the presence of a particular application on a host h_1, we know that information about running applications is generated only by host h_1. Using this locality rationale, we identify a portion of each rule. The exe-

cution of this portion that considers only local statements on each agent is equivalent to an execution that considers all statements in the system. Such a portion is executed independently on each agent where the state information is acquired, and only the results are shared with the rest of the system.

Our validation process is composed of two phases: decomposition and execution. The decomposition phase uses the RDA algorithm to integrate information about the locality of agents' configuration statements with the rules of the organization. The result of this process is a decomposition of each rule into *local sub-rules* and *aggregate sub-rules*. In the execution phase, monitoring agents use local sub-rules to perform partial processing of the rule and use predicate groups to send processed state information. Verifiers use aggregate sub-rules to control the process of aggregating information acquired from multiple agents.

Decomposition

The decomposition phase takes a rule and information about the statements generated by agents to decompose the rule into local and aggregate sub-rules. This process uses an RDF-graph representation of the rules which is more suitable for our analysis. Each rule is transformed in a *rule graph*, a directed multigraph describing the explicit relationship between variables, resources, and predicates used in the rule. The graph has a node for each resource or variable and an edge from subject to object for every statement pattern.

Definition 3. *Given a rule R composed of a set of resources and variables N and a set of statement patterns S, the* **rule graph** *G_R is an directed multigraph $G_R = (N, S, s, o)$, where N and S are the resources and statement patterns used in the rule, $s : S \to N$ is a function mapping each statement pattern to the subject of the statement, and $o : S \to N$ is a function mapping each statement pattern to the object of the statement.*

The statement pattern h defined in the rule head is marked as the *head edge*. The conversion of the rule in Example 1 into a rule graph is shown in Figure 4.3.

Locality: For an agent a, we say that a statement pattern is *local* if all its potential matching statements are always found locally, independently from the current state of the system. For identifying the local portion of the

rule, we formalize our knowledge about the locality of the statement patterns using a RDF graph we call the *locality graph*. One of the nodes of the graph, called the *anchor*, identifies a specific type of agent as the information source (e.g., Host).

Definition 4. *A* **locality graph** *is a directed multigraph* $L = (N, S, s, o)$ *where* N *is a set of variables or resources,* S *a set of statement patterns, and* s *and* o *defined as before. A specific element* $s \in N$ *is defined as* **anchor node**.

The locality graph depends on the structure of the monitoring system and it is defined at design time. To verify if a statement pattern is in the locality graph, we start from the anchor node. We substitute the anchor node value with the name of the local resource l (e.g., if the local resource is an host, we take as a value of the anchor node the name of the host). A statement pattern (l, p, o) is part of the locality graph if all events matching the given pattern are generated locally on the agent a.

Example 3. *For the rule in Example 1, if we consider a monitoring system able to obtain information about the servers running on a host, the event pattern* $(S, provides, A)$ *is part of a locality graph defined on an anchor node* S *of type Host. This can be checked by considering that, given a host* h_1*, all event patterns* $(h_1, provides, A)$ *can be found locally: the monitoring system is able to provide information about all services running on a generic machine* h_1*. Similarly, we can consider the statement pattern* $(S, istypeserver, true)$ *as part of the locality graph.*

The notion of locality can be generalized to paths in the graph. Each undirected path starting from the anchor node represents a conjunction of statement patterns: all the statements matching such a combination of patterns are found locally on each agent.

Example 4. *If we continue to analyze our example, we can consider the predicate pattern* $(A,\ hasvuln,\ V)$*. Now, we consider a monitoring system on the agent that queries the NVD database about its running programs to identify vulnerabilities. Hence, if a program is vulnerable, the local agent is able to identify the vulnerability. Because of this new local knowledge, we can consider the conjunction* $(S, provides, A) \wedge (A, hasvuln, V)$*. For a host* h_1*,*

all events $(h_1, provides, A)$ *are local and, for each local program* a_1*, we have that* $(a_1, hasvuln, V)$ *is found locally. Hence, we can add to the locality graph an edge* $(A, hasvuln, V)$*. Similarly, we can consider the statement pattern* $(A, istypeservice, true)$ *as part of the locality graph, as an agent can identify that a resource A matching the previous constraints is a service. The locality graph for this example is shown in Figure 4.4(a).*

Locality graphs are constructed at design time and rarely change: they depend on the capability of the monitoring system. A locality graph needs to be updated only when new monitoring capabilities are added. Statements used in the validation of critical policies should not be part of the locality graph.

Using the locality graph we can identify subgraphs of the rule graph which can be processed locally on the agent collecting data. For clarity, we consider only one type of anchor, `Host`. We consider a rule graph G_R. For each resource r of type Host, we generate a *agent-local* subgraph as follows. As r corresponds to the anchor node of the locality graph, we can start a recursive visit of the locality graph. If an edge appears both in the rule graph and in the locality graph, the statement pattern associated with the edge might be processed locally and it is added in the agent-local subgraph (we use Datalog unification to match edges when they contain variables). Once the visit to the graph is completed, each agent-local subgraph contains a set of statement pattern that can be completely validated locally, for a given host.

Example 5. *Consider our locality graph containing the nodes* $N = \{S, A, V, true, true\}$ *and the edges* $(S, provides, A)$*,* $(A, hasvuln, V)$*,* $(A, hasvuln, V)$*,* $(S, istypeserver, true)$*,* $(A, istypeservice, true)$*. We can consider the rule graph associated with our the rule in Example 1. The processing starts by taking all nodes of type Host—S, in our case. We recursively visit the locality graph and we see that all edges* $(S, provides, A)$*,* $(A, hasvuln, V)$*,* $(S, istypeserver, true)$*, and* $(S, istypeservice, true)$ *are contained both in the locality graph and in the rule graph. For this reason, they are added to an agent-local graph. The agent-local graph for our example is shown in Figure 4.4(b).*

Without loss of generality, for every agent we choose one of the agent-local subgraph to be *local*. Certain agent-local subgraphs could have anchors

which are not local for an agent. For example, given a locality graph (H, p, A) and a rule $(h_2, p, A) \rightarrow (h_2, violation, A)$, the subgraph is local only for host h_2. Such kind of agent-local subgraphs can be considered *local* only in the appropriate agents.

Transformation into sub-rules: Once the local subgraph is identified, we generate local and aggregate sub-rules to use for the distributed rule processing. These sub-rules specify the location of the computation for the validation of rules and the structure of the communication between agents and verifiers.

A sub-rule is a pair $< \beta \rightarrow \eta, \mu >$ formed by a rule $\beta \rightarrow \eta$ (β is the body, η the conclusion) and a query μ. For local sub-rules, the rule $\beta \rightarrow \eta$ represents a portion of the original rule, and the query μ identifies the statements generated by local processing which are sent to verifiers. For aggregate sub-rules, the query identifies the information received by the agents, and the rule identifies the remaining portion of the policy validation.

Local sub-rules: For each rule graph we consider its local subgraphs. There are several cases. (i) If the local subgraph covers the entire rule $\beta \rightarrow \eta$, then we create a sub-rule $< \beta \rightarrow \eta, \emptyset >$. In this case, the entire processing of the rule can be local and the agent only needs to raise an alarm for violations. (ii) If only the head η statement of the rule graph is not part of the local subgraph, we create a local sub-rule $< \beta \rightarrow \eta, \eta >$. i.e., we locally compute the entire rule, and we share with the verifiers only the consequences. (iii) If the local subgraph covers only a portion of the rule, then we create several local sub-rules. For each edge π of the rule graph not in the local subgraph, we create a local sub-rule $< \emptyset, \pi >$, and we generate a single local sub-rule $< \beta_l \rightarrow \eta_l, \mu_l >$ for the entire local subgraph as follows.

The body of the rule β_l is constructed by taking all edges in the local subgraph and converting them into a conjunctive query of predicate patterns. Because all edges are part of the local subgraph, all statements that match the body of the rule are found locally. For example, each match of a rule body $(A, p_1, B), (B, p_2, C)$ creates a tuple (A, B, C) containing the values of the variables. These tuples need to be shared with the verifiers to complete the validation of the rule. However, we do not need to share the entire set of variables, but only the variables which are used in statement patterns outside the local subgraph. Their variables are identified by considering the nodes at the boundary of the local subgraph (i.e., nodes which have an edge in the

local subgraph and an edge outside it). For example, if only the variables A and C are used in statements outside the local subgraph, we only need to share the tuple (A, C).

This tuple, which represents the information to share, is used as the head η_l of the rule. However, because the size of the tuple can change depending on the number of variables in the body, we represent the head using a variable number of statements which share the same RDF blank node as the subject. We can think of blank nodes as resources with a unique name generated automatically by the system. We identify an RDF black node with a resource name starting with the character _ : (e.g., _ : k is a blank node). The object of each statement is one of the variables in the body, and the name of the predicate depends on the rule and on the name of the variable. We call these statements *rulematch* statement patterns and they are identified with a predicate $rm_{r,v}$ where r is an id of the rule, and v is the name of the variable to which they refer.

Example 6. *Continuing with Example 1, the locality graph only covers a portion of the rule. The local subgraph is converted into a single local sub-rule. The body of the rule is* $(S, istypeserver, true)$, $(S, provides, A)$, $(A, hasvuln, V)$, $(A, istypeservice, true)$. *The head of the rule is composed of a set of rule match statement, one for each variable that used in the rest of remaining part of the rule:* $(\_ : k, rm_{r,'S'}, S)$, $(\_ : k, rm_{r,'A'}, A)$. *The blank node* _ : k *is substituted with the same random string for all statements,* r *is a unique identifier of the rule and* $'A'$ *and* $'C'$ *are strings representing the variable names. The head of the rule is also the query* μ *for the local sub-rule.*

The non-local statements in the rule are converted into local sub-rules with an empty rule. In particular, we obtain the following queries: (S, *criticality*, *high*), (S, *hasexception*, *true*). *The conversion result is shown in Figure 4.4(b).*

The body β_l of the local sub-rule represents only a portion of the body of the original rule. Because of this, there are corner cases when the β_l might not be "safe" as the statement patterns containing the non-negated variable in the original "safe" rule might not be included in the local rule. If, for the rule under analysis, such a condition occurs, we consider only a reduced subgraph that does not contain the edges covering the unsafe portion of the rule and any subsequent edges so that the local β_l is always safe.

Aggregate Sub-Rules: The analysis for the generation of aggregate sub-rules is similar to the generation of local sub-rules. Even if aggregate sub-rules are executed on verifiers, we still use the concept of "locality" as locality for the agents. For edges $\pi = (A, p, B)$ not in the local subgraph we create an aggregate sub-rule with only a query $< \emptyset, \pi >$. This aggregate sub-rule specifies that all statements matching this pattern should be delivered to the verifier. If the rule graph ρ does not have a local subgraph, we add an aggregate sub-rule $< \rho, \emptyset >$ which introduces the rule in the verifier's KB. Hence, for rules with no local subgraphs, the verifiers simply collect all statements matching any of the statement patterns of the rules and performs local reasoning.

For rule graphs with a local subgraph, we need to account for the partial processing performed on the agents. We create an aggregate sub-rule with a rule ρ' where the local subgraph edges have been substituted with rulematch statements, and we create a set of queries that collects such statements.

Example 7. *Continuing with Example 1, we account for the local processing performed on the agent and we obtain the aggregate sub-rule* $(\_ : k, rm_{r,'S'}, S)$, $(\_ : k, rm_{r,'A'}, A)$, $(S, criticality, high)$, $\neg(S,\ hasexception,\ true) \rightarrow (r_1,$ *violation*, $V)$.

Execution

The execution is driven by local sub-rules and aggregate sub-rules. For each aggregate sub-rule $< \rho, \mu >$ the verifier adds the rule ρ to its local KB. On the agents, for each local sub-rule $< \rho, \mu >$ we add the rule ρ to the local KB and we select all statements matching μ. The execution phase is based on the predicate group mechanism introduced in Section 4.6.3. Agents join the predicate groups based on the predicates defined in the queries of the rewritten ruleset, and verifiers use the body of the generated rule-sets to select the predicate groups to use for receiving updates.

For the validation of critical policies, we require verifiers to collect statements from a minimum number of different agents. A Byzantine agreement on the result is reached by broadcasting the result of local validations to other verifiers.

The DHT infrastructure supports the monitoring for liveness. Each mon-

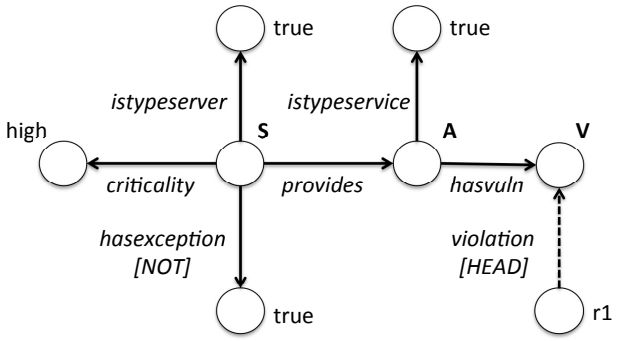

Figure 4.3: Conversion of the rule of Example 1 into a rule graph.

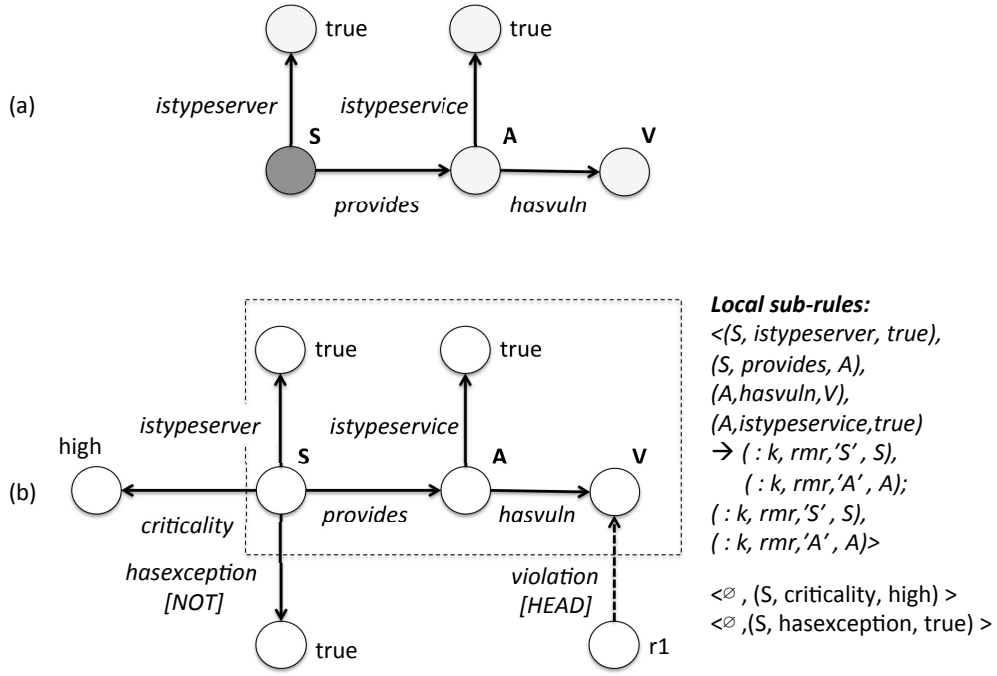

Figure 4.4: Example of the conversion of a rule graph into a set of local sub-rules. (a) Locality graph. The S node (in dark grey) is the anchor node of the graph. (b) Rule graph annotated with locality information and its conversion into a set of local sub-rules.

itoring agent is registered to several predicate groups, and the agents periodically send heartbeat messages to their neighbors. When the failure of an agent is detected, a message is sent to the registered verifiers so that all statements that had been generated by the failed agent are removed from the verifiers' state. As each host is registered to several trees, even the failure of all the hosts in a branch of the tree are detected by the parent hosts in other trees.

Proof of Correctness

The proof of correctness of our algorithm requires the definition of a formal model of our monitoring system. The state of the system that we are interested in monitoring is represented as a knowledge base $\mathcal{G}$ that contains all the known facts about the system. During the execution, such knowledge is distributed across the monitoring agents $\mathcal{L}_i$. Verifiers place data they receive from the agents in a set of aggregated knowledge bases $\mathcal{V}_i$. For simplicity, we assume a uniform set of monitoring agents with the same locality graphs and we assume a single verifier $\mathcal{V}$ that aggregates data from the agents.

The goal of a compliance monitoring system is to detect all policy violations present in the known state of the system. The global verification for the presence of policy violations is performed by analyzing the presence of statements indicating violations in a knowledge base $\mathcal{G}^{+_R}$ obtained by applying consequence finding to the knowledge base $\mathcal{G}$ using the global set of rules R. In our system, we avoid performing such a centralized inference by distributing the computation across the distributed knowledge bases managed by the monitoring agents ($\mathcal{L}_i$) and by the verifiers ($\mathcal{V}_i$). The proof of correctness needs to show that the distributed inference performed using our rule decomposition algorithm finds all violations that are found by the application of the original set of rules on the global state $\mathcal{G}$.

The proof is composed of four parts. First, we show that if we use the RDA algorithm to decompose a ruleset R into the two sets R_L and R_G, the ruleset $R_L \cup R_G$ preserves the original properties of safeness and of stratified negation. Second, we show that $R_L \cup R_G$ is equivalent (for our purposes) to R. We represent statements indicating a violation with a pattern n (e.g., $(*, violation, *)$) and we define an operator of selection σ_n that selects only statements matching the pattern n. R is equivalent to $R_L \cup R_G$ if the two rule-

sets used over the same knowledge base find the same set of violations. Formally, we show that $\sigma_n \mathcal{G}^{+_R} = \sigma_n \mathcal{G}^{+_{R_L \cup R_G}}$. Third, we show that the execution of the local rules on each local knowledge base is equivalent to the execution of the local rules on the global knowledge base. Hence, $\bigcup \left(\mathcal{L}_i^{+_{R_L}} \right) = \mathcal{G}^{+_{R_L}}$. Last, we show that by creating a knowledge base $\mathcal{V}$ by filtering predicates $\mathcal{V} = \bigcup \sigma_V \left(\mathcal{L}_i^{+_{R_L}} \right)$, we have that $\mathcal{V}^{+_{R_G}} = \mathcal{G}^{+_R}$. This last step provides the complete proof of the algorithm.

First, we prove that the new set of rules $R_L \cup R_G$ preserve the original properties of safeness and stratified negation of the original set R.

Lemma 3. *Given a set of rule R which is safe and is with stratified negation and two sets of rules R_L and R_G generated using the RDA algorithm, we have that:*

- *R_L is safe*
- *R_G is safe*
- *$R_L \cup R_G$ is safe and is with stratified negation*

Proof. To prove such conditions we analyze the operations of the algorithm. We show that every rule $r_l \in R_L$ is safe by construction by looking at the three cases considered in the algorithm: 1) the rule is entirely local; 2) the rule has a local body and a global head; and 3) the body of the rule is partially local. For cases 1) and 2), the body β_l of the sub-rule is equal to the body β of the original rule, hence β_l is safe as β is safe. In 3), the algorithm reduces the local portion of the rule until a safe rule is found. Hence, it is safe by construction. Each rule $r_g \in R_G$ is obtained by substituting a subset of the original rule r with rulematch statements. The rulematch statements are non-negated statement patterns that contain all the variables shared between the considered subset of the statement patterns and the remaining part of the rule. As the substitution only adds non-negated statement patterns and the shared variables are preserved, the rule we obtain is still safe.

Last, we show that $R_L \cup R_G$ is safe and is with stratified negation. Safety immediately follows from our previous result, as every rule in the set is safe. As the rewriting does not add negated statements, the rewrite still preserves the same stratification of the rules. Hence, as the original ruleset has stratified negation, the resulting ruleset preserves the same property. □

Second, we show that $R_L \cup R_G$ and R identify the same set of violations.

Lemma 4. *Given a set of rules R and two sets of rules R_L and R_G obtained with the RDA algorithm, we have that $\sigma_n \mathcal{G}^{+_R} = \sigma_n \mathcal{G}^{+_{R_L \cup R_G}}$.*

Proof. We prove this lemma by contradiction. First, we show that if a violation v is present in $\mathcal{G}^{+_R}$, then v is also present in $\mathcal{G}^{+_{R_L \cup R_G}}$. We assume that there exists a violation v in $\mathcal{G}^{+_R}$ that does not appear in $\mathcal{G}^{+_{R_L \cup R_G}}$. Because v is true, there must exists a rule $r : \beta \rightarrow v$ in R so that there exist an assignment θ of the variables in the body of β that makes each statement pattern in β match with a statement in $\mathcal{G}$ (or not match, if the statement pattern is negated). On $R_L \cup R_G$, it can be the case that r is preserved during the rewrite and placed in either R_L or R_G. In such a case, it is trivial to see that the same assignment of variables would make v true in $\mathcal{G}^{+_{R_L \cup R_G}}$, hence leading to a contradiction. A more interesting case is when the rule r is rewritten in a local rule $(\beta_l \rightarrow p)$ and a global rule $(p, \beta_g \rightarrow v)$. However, even in such a case we have a contradiction. The assignment θ would also make all the statement patterns in β_l and β_g be true. As we have the rule $\beta_l \rightarrow p$, we can infer that p is true. Hence, the entire body of the global rule $(p, \beta_g \rightarrow v)$ is true, from which we can infer that v is true and reach a contradiction.

The opposite direction of the implication is verified in a similar way. If we consider a violation $v \in \mathcal{G}^{+_{R_L \cup R_G}}$, then we have a substitution θ_g that makes true every statement pattern in the body of the rule $(p, \beta_g \rightarrow v)$. As p is generated only by a rule $r_l \in R_L$, there must exist another substitution θ_l so that the body β_l of a rule $(\beta_l \rightarrow p)$ is true. Because p contains all the variables shared between β_g and β_l, we can create another substitution $\beta = \beta_l \cup \beta_g$ without conflicts on the values of the variables. As the original rule in r is in the form $(\beta_l, \beta_g \rightarrow v)$, we have that the substitution β makes the body of the rule r true. This fact leads to a contradiction and proves the lemma. □

Third, we prove that performing local reasoning using R_L independently on each of $\mathcal{L}_i$ is equivalent to using R_L on the global knowledge base $\mathcal{G}$.

Lemma 5. *We consider a system composed of a set entities a_i (identified by anchor resources) able to perform local reasoning. Each anchor resource a_i is associated with a knowledge base $\mathcal{L}_i$. Given a set of rules R_L created using*

the RDA algorithm and the completeness graph $\mathbf{C}$*, we have that* $\bigcup \left(\mathcal{L}_i^{+_{R_L}} \right) = \mathcal{G}^{+_{R_L}}$.

Proof. We consider a rule $r_l \in R_L$ created from C and an anchor resource a_i. Becase the rule $r_l : \beta_l \rightarrow p$ is local, it has a variable A that is associated with the anchor resource a_i. Hence, we can create a substitution $\theta_a = A/a_i$. By definition of the completeness graph, we know that if there is a set of statement patterns β_c that can be generated with a path in the graph $\mathbf{C}$ and a substitution θ_a, any set of statements matching $\theta_a(\beta_c)$ is going to be found locally on $\mathcal{L}_i$. As the body β_l of the rule is a generated from a subgraph of $\mathbf{C}$, we have that $\mathcal{G}^{+_{\theta_a(\beta_l \rightarrow p)}} = \mathcal{L}_i^{+_{\theta_a(\beta_l \rightarrow p)}}$.

Given these conditions, we consider each direction of the implication. First, given a statement p and a rule $r_l : \beta_l \rightarrow p$, we show that if $p \in \mathcal{G}^{+_{r_l}}$ then we have that $p \in \bigcup \left(\mathcal{L}_i^{+_{r_l}} \right)$. We can consider a substitution θ_g that makes the β_l true in $\mathcal{G}$. Because β_l is a subset of $\mathbf{C}$ rooted by an anchor resource, we have that θ_g must contain a substitution $\theta_a = A/a_i$, where a_i is an anchor resource. As each anchor resource is associated with a knowledge base $\mathcal{L}_i$, we have that all statements matching $\theta_a(\beta_l)$ are found locally on $\mathcal{L}_i$. Hence, there exist an $\mathcal{L}_i$ on which p is proven as true. As the head of a rule r_l is always a non-negated statement, we have that the union over all $\mathcal{L}_i$ preserves the truth of the statement. Proving the other direction is simpler. $\mathcal{G}$ is, by definition, composed of the union of all $\mathcal{L}_i$. By our problem statement, we assume that there is no conflicting information contained in $\mathcal{L}_i$ (such conflicts can indicate security problems and are managed through a different process that selects the valid information to use). Hence, if there exist a substitution θ_g that makes a rule β_l true in one of the $\mathcal{L}_i$, the substitution also makes β_l true in $\mathcal{G}$. □

Finally, we show the overall correctness of the algorithm.

Theorem 4. *We consider two rulesets* R_L *and* R_G *generated using RDA from a ruleset* R*, a set of local knowledge bases* $\mathcal{L}_i$*, and the global knowledge base* $\mathcal{G}$*. Also, we consider a knowledge base* $\mathcal{V}$ *obtained by acquiring from the local agents the statements that have predicates matching any of the predicates in the rule body of the rules in* R_G *(i.e., we define a query* V *is a query obtained by putting in OR all statements patterns in any of the rules, and we define* $\mathcal{V} = \bigcup \sigma_V(\mathcal{L}_i^{+_{R_L}})$*). We have that* $\mathcal{V}^{+_{R_G}}$ *is equivalent to* $\mathcal{G}^{+_R}$

Proof. First, we substitute the knowledge base $\mathcal{V}$ with its definition. In this way, we rewrite the previous formula as $\bigcup \sigma_V(\mathcal{L}_i^{+_{R_L}})^{+_{R_G}}$. Using Lemma 5 we rewrite again the expression as $\sigma_V(\mathcal{G}^{+_{R_L}})^{+_{R_G}}$. σ_V selects, by construction, all statement patterns that are used in the body of any rule in R_G. As statement patterns not selected by σ_V do not have any effect on the final inference, we can simplify the expression as $(\mathcal{G}^{+_{R_L}})^{+_{R_G}}$.

We show that by applying the inference rules in R_L and then, on the resulting sets, applying the rules defined in R_G is equivalent to applying the ruleset obtained by $R_L \cup R_G$. This is not true in the general case: the consequence of a rule in R_G might be used in the body of a rule in R_L. Applying the inference sequentially would lead to an incorrect inference. However, in our case, we have that the body of every rule in R_L can only match statements generated locally by a device (by construction). It cannot contain any statement inferred from information in the global state, as such information would not be local and, hence, not be part of the completeness graph. Because of this, we can rewrite the expression as $\mathcal{G}^{+_{R_L \cup R_G}}$. By applying Lemma 4, we prove our theorem. □

CHAPTER 5

RESOURCE-BASED APPROACHES

In infrastructure monitoring systems, policies describe conditions on the state of resources such as computer systems, networks, or users. A violation is identified when a limited set of resources violates such conditions. Our second theme for the definition of monitoring algorithms is based on the intuition that events taking part in a policy violation are related to each other as they provide information about a limited set of resources that, together, create the violation of the policy. For example, we can consider a policy that specifies that a user in a computer lab should not be logged on two machines at the same time. A monitoring system could generate an event when a user logs into a machine and when the user logs out. A violation would be represented by two logins of the same user on different computers. The two events, if creating a violation, are related to each other as they provide information about the same user. Our system finds violations through the identification of a sequence of events that corresponds to incorrect or invalid changes in the state of such resources. We split the problem into a set of steps. At each step, we find events related to a single resource that is potentially involved in the violation. Using the relation between resources, we integrate such partial sequences with events involving other resources to reconstruct the complete sequence of events in the violation.

This chapter describes two approaches based on resource-based sharing. Both approaches reduce the information shared across domains and require information sharing only among servers managing resources potentially related in a violation. The first part of the chapter describes a rule-rewriting approach that distributes the computation across a set of peer monitoring servers managing different resources. For each resource, the system defines a single server managing it. While different servers can manage different resources, a resource is always managed by a single server. All events related to such a resource are aggregated in such a server which checks if the state

of such a resource can potentially contribute to a violation. If it is the case, information about such an occurrence is shared with servers managing another resources potentially involved in the same violation, until a complete violation is found.

Aggregating all events about a resource in a single monitoring server can itself be challenging. It is often the case that the entire state of a resource is not known by a single domain. For example, in a cloud computing scenario, information about a virtual machine instance is partially collected by the cloud provider (e.g., placement of the virtual machine, physical network connections, relation with other user's virtual machines) and partially by the cloud user (e.g., running software, application configurations). Aggregating information in a single server under the control of either of the two domain would create a large sharing of information not contributing to violations. For this reason, our second approach removes the requirements that all data about a resource be collected in a single server. In such an approach, multiple monitoring servers can maintain partial information about resources. We introduce a distributed algorithm for correlating such data. Our solution takes advantage of cryptography techniques to reduce the amount of information shared at the minimum need-to-know for simple policies.

5.1 Distributed Detection of Policy Violations

In our first approach, we define a distributed algorithm for the detection of policy violations. The algorithm defines both where information is stored in monitoring servers, and what information is exchanged among the servers for detecting all violations.

We distribute the detection of violations across a set of peer monitoring servers that we call brokers. In such an architecture, monitoring servers within each organization receive information about the resources within their system. To map uniquely each resource to a specific monitoring server, we use a hash function that takes a resource name and maps it to a single monitoring server across the entire system. When multiple organizations are involved, such a function attempts to respect domain boundaries by assigning resources managed within each domain to one of the monitoring systems within the same domain.

Our algorithm validates compliance by aggregating events in a series of steps. Each step correlates events related to a single resource. If the single-step correlation identifies that the resource can potentially contribute to a violation, we send a summary to the next step so that it can be correlated with additional resources.

We distribute resources across brokers using the hash function. If the entire policy is defined on a single resource r, then the policy is validated completely in the broker managing r. Events about r are directly delivered to the broker and inserted into a local knowledge base. For example, we can take the resource $host_a$ (associated to a broker x) and the policy $\texttt{critical}(A)$, $\texttt{poweroff}(A) \rightarrow \texttt{violation}$. The two events $\texttt{critical}(host_a)$ and $\texttt{poweroff}(host_a)$ are delivered to the broker x and inserted in its local knowledge base. The knowledge base matches both events and finds a violation. It is easy to prove that if two single-resource events are correlated by a policy, both events are always sent to the same broker.

When events are related to multiple resources, the problem becomes more complex. For example, a policy that relates IDS events with the presence of vulnerable programs running on a computer system requires integrating events about several resources such as computer systems, software, and specific network flows. In general, events involved in a violation cannot be related to a single resource. For example, an event could state that a computer system $host_a$ is running a program $\texttt{runs}(host_a, p)$ that is untrusted $\texttt{untrusted}(p)$. Another event could specify that a system $host_b$ is critical for the system $\texttt{critical}(host_b)$. A third event could state that there is a connection between $host_a$ and $host_b$, $\texttt{connection}(host_a, host_b)$. Even if these events are related to each other, no single resource is shared across all of them.

5.2 Rule Rewriting

Our algorithm performs the distributed resource-based correlation by rewriting monitoring rules. We analyze a policy and create an equivalent set of rules that we call *resource ruleset*. Each rule requires information about a single resource and represents a partial violation of the policy. If a partial violation of the policy is found, a new event is generated. We forward the event to the broker managing one of the resources connected to the potential violation.

For example, a policy could specify that there is a violation if a machine connected to an internal network receives a connection directed to a vulnerable program. We represent this policy as $\texttt{connected}(H, N)$, $\texttt{internal}(N)$, $\texttt{runs}(H, S)$, $\texttt{vulnerable}(S)$, $\texttt{conn}(H, S, IP) \rightarrow \texttt{violation}(H, S, IP)$. We can rewrite the policy as follows:

$$\begin{aligned} &\texttt{runs}(H, S), \texttt{vulnerable}(S) \rightarrow \texttt{partial}_S(H, S) \\ &\texttt{internal}(N), \texttt{connected}(H, N) \rightarrow \texttt{partial}_N(H) \\ &\texttt{partial}_S(H, S), \texttt{partial}_N(H), \texttt{conn}(H, S, IP) \rightarrow \\ &\qquad\qquad\qquad\qquad \texttt{violation}(H, S, IP) \end{aligned} \tag{5.1}$$

The first rule relates events about the resource identified by the value of the variable S. The second rule relates events about the resource identified by the value of the variable N. The third rule relates the partial information from the previous rules with events about the resource H. The conclusions of the first two rules are events of type `partial` representing a partial processing of the original rule. This new set of rules generates the same statements $\texttt{violation}(H, S, IP)$ as the original statements, but it is formulated as a sequence of rules that filter events in different steps.

As the body of each new rule relates events about a specific resource, we can find all of its conclusions by aggregating in the same broker all events about such a resource. For example, if the two events $\texttt{internal}(net_1)$ and $\texttt{connected}(H, net_1)$ (for all H) are sent to the broker associated with the resource net_1, such a broker is able to compute all the couples of resources (net_1, H) that match the body of the rule. We summarize the partial evaluation with the statements $\texttt{partial}_N(H)$. Changing the value net_1 (i.e., value of the variable N) changes the broker in charge of the correlation. For example, if we consider $N = net_2$, the two events $\texttt{internal}(net_2)$ and $\texttt{connected}(H, net_2)$ are sent to a different broker associated with the resource net_2. The partial conclusions computed by each broker are forwarded to the brokers managing the resource identified by the value of the variable H of $\texttt{partial}_N$. On the brokers associated with the different values of the variable H, we perform the same process and we integrate the partial events $\texttt{partial}_N$, $\texttt{partial}_S$, and the event `conn`. Using our algorithm, brokers share data only if such an interaction is necessary for detecting a possible violation.

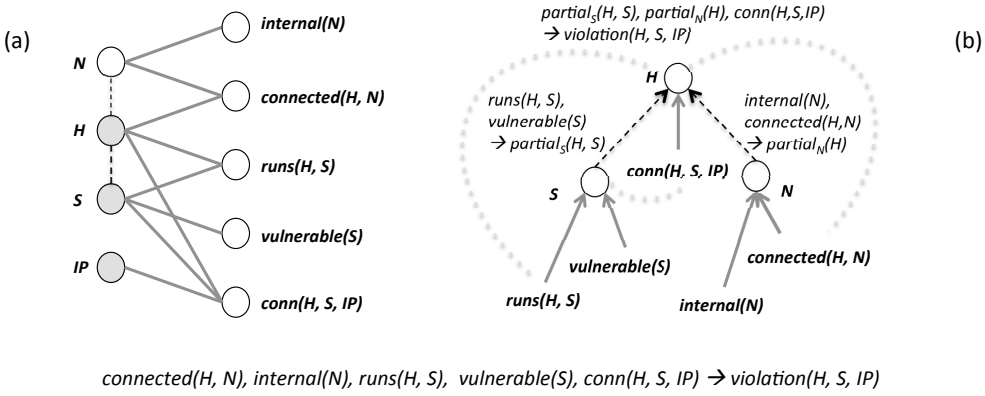

Figure 5.1: (a) Policy graph. Dashed lines are added during the conversion process; (b) Policy tree generated by choosing H as root. Dotted lines are graph edges removed during the conversion process.

5.3 Distributed Correlation Algorithm

The process of rule rewriting starts with the identification of the resources used in the rule. We represent this information in a *policy graph*. The policy graph is a bipartite graph where one set of nodes represents variables and constants used in the rule, and the other represents predicates. We create an edge between a predicate node and a variable node if the predicate contains the variable. A policy graph for an example policy is shown in Fig. 5.1(a).

The policy graph shows the relation between variables and events. If the graph is not connected we modify the policy by adding a common fictitious resource that connects the two parts of the policy. However, we believe that unconnected polices are rare as they would be the equivalent of a SQL join of two tables without a "where" condition.

Using the graph, a policy is rewritten as a resource ruleset in two steps. First, we choose an order for the correlation steps by computing a spanning tree of the graph that we call *policy tree*. Then, we generate the resource ruleset from the tree. As the root of the spanning tree represents the final step of the correlation process, we choose the root to be one of the resources that appear in the final violation notification message, so that information about such a resource is readily available for creating the notification. Fig. 5.1(b) provides an example of the conversion of a policy graph to a policy tree. The steps necessary for such a conversion are as follows.

1. We remote variable nodes connected to only one predicate node. In Fig. 5.1(a), we remove the node IP.

2. We explicitly represent the relations between resources by adding edges between variable nodes. We create a new edge between variable nodes of distance 2 (i.e., we ignore predicate nodes and we connect directly neighboring variable nodes). In our example of Fig. 5.1(a), we mark such edges with dashed lines.

3. We compute the spanning tree by selecting a root and by removing redundant edges. For each predicate node, we maintain only one edge. We select the edge to ensure that each variable node is connected to at least one event containing a reference to the variable node's parent. In Fig. 5.1(b), we mark the removed edges with dotted lines.

This transformation connects each predicate node (i.e., events) to a variable representing one of the resources contained in the event. Events are forwarded to the broker managing the resource represented by the value of the variable. For example, we can take a predicate $\texttt{connected}(H, N)$ with an edge to N. An event $\texttt{connected}(host_1, net_1)$ is forwarded to the broker managing net_1. For the rest of the section we indicate with $\mathcal{N}$ the resource associated with the value of the variable N.

A broker managing a resource $\mathcal{N}$ can perform a partial matching of the policy. For each variable node, we create a rule of the resource ruleset by considering the incoming edges. At the lower heights of the tree, all incoming edges of a variable node are connected to predicate nodes. These predicate nodes all share the same variable H. For example, the body of the rule associated with the variable node N in Fig. 5.1(b) is $\texttt{internal}(N)$, $\texttt{connected}(H, N)$. For a violation of the policy to exist, the variable N must have the same value on all predicates contained in the policy. As the broker managing $\mathcal{N}$ receives all events that share the same value for the variable, the partial matching is complete. The head of the partial rule is a predicate that has the variables used in the partial rule as parameters. In this case, the result is encoded with $\texttt{partial}_N(H, N)$. This predicate is treated as a new event, and it is forwarded up in the tree. In our example, the event is sent to the broker $\mathcal{H}$. Step 3 in the conversion of the graph to the tree ensures that the broker always has at least one event that carries both the information about the current resource and about the resource one level higher in the tree.

A variable node can have incoming edges connected to other variable nodes. These edges are considered as connected to events representing the partial execution. In our case, the node H is associated to the rule $\texttt{partial}_S(H, S)$, $\texttt{conn}(H, S, IP)$, $\texttt{partial}_N(H, N)$. If the variable node is the root, the rule is directly associated with the presence of a violation. When multiple policies are present, each policy uses a different name for the partial execution predicates.

During the execution of this algorithm there is no single broker managing the resolution for an entire policy: depending on the resources mentioned in the events, different brokers are in charge of the different steps of the aggregation and of the final identification of violations.

5.3.1 Remove unnecessary information

We remove unnecessary data from partial execution statements to reduce the amount of information sent to other brokers. If a variable is not used anywhere else in the rule, we can drop it from a partial statement without affecting the equivalence to the original policy. For each variable node V, we build a set of variables we call *shared set*. The shared set is constructed by removing the subtree of V from the policy tree and by taking all variables used in the remaining tree. We drop from the partial execution statements of V the variables that do not appear in the shared set.

5.3.2 Distribute information about critical resources

A pure resource-based distribution of events offers little protection against attackers interested in acquiring information about a specific critical resource in the system. An attacker can acquire these events by identifying the broker managing such a resource and by compromising it. Our algorithm provides a protection against this type of attack by spreading events about critical resources across multiple brokers. As different types of events about the same resource are generally used in different policies, we add a policy-dependent prefix in the selection of the broker. When a resource is identified as critical, instead of relying only on the resource name r for the identification of the broker $H(r)$, we add a prefix p_i that depends on the policy in which the

resource is used. Different types of events about a critical resource are sent to different brokers $H(p_i|r)$.

We consider critical all resources representing data about the monitoring servers. In this way, we limit the possibility that an attacker could use information collected from the compromise of a monitoring server to compromise another server.

5.4 Correctness Argument

Our algorithm is correct and complete if it identifies the same set of violations that would be identified if all events were integrated in a global KB. We show the equivalence in two steps. First, we show that the processing of each rule in the resource ruleset on a broker is equivalent to the processing of the same resource-ruleset rule in a global knowledge base. Second, we show that the combination of the resource ruleset is equivalent to the original rule.

All predicates in a resource-ruleset rule share a common variable V. For any instantiation $V = v$, the rule is triggered only if events have the same value v for the variable V. Our algorithm ensures that if a predicate p is part of the resource-ruleset rule for a variable V, all events having $V = v$ are sent to the same broker. Hence, such a broker can identify all inference for which $V = v$. As the same rule is repeated in all brokers, we obtain the same result for every value of V.

Second, we show the equivalence between the resource ruleset and the original rule by noticing that the rewrite algorithm is, in fact, a simple logical manipulation of the rule. The rule-generation algorithm can be seen as a set of rewriting steps that preserve the equivalence. We choose a variable V_1 and create a rule $rule_{V_1}$ that has a body containing all the predicates $\texttt{pred}_i(\overline{A_i})$ (where $\overline{A_i}$ is a set of variables) such that $V_1 \in \overline{A_i}$. The head of the rule is a new predicate $\texttt{partial}_{v_1}(\overline{A_{V_1}})$ where $\overline{A_{V_1}} = \{\cup\overline{A_i}|V_1 \in \overline{A_i}\}$. We remove the selected predicates from the body of the original rule and we substitute them with the $\texttt{partial}_v(\overline{A_v})$. For each $rule_v$, because the predicate $partial_v$ is unique for the rule, we have that a particular assignment of $\texttt{partial}_{V_1}(\overline{A_{V_1}})$ is possible if and only if all predicates $\texttt{pred}_i(\overline{A_i})$ in the rule body are also true. Hence, at the end of the rewrite process, the initial rule has been rewritten as $\texttt{partial}_{V_1}(\overline{A_{V_1}}), \ldots, \texttt{partial}_{V_k}(\overline{A_k}) \rightarrow \texttt{violation}(\overline{A})$. Because each of the

$\texttt{partial}_v$ is true if one only if the body of its rule is true, and $\overline{A_{V_1}}, \ldots, \overline{A_{V_k}}$ still represent the entire set of variables, we have that $\texttt{violation}(\overline{A})$ is generated if and only if it is generated by the original rule.

5.5 Garbled Circuit-based Processing

Our second approach generalizes our previous approach by permitting each organization to declare the resources managed by each monitoring system, and improves it by introducing a cryptography-based mechanism for reducing the amount of information shared across domains [70]. Our distributed algorithm identifies portions of such event sequences within each domain, and uses a distributed naming system to identify monitoring systems potentially containing other portions of the event sequence causing a violation. We use secure two-party computation to connect such partial sequences across domains, so that we can identify the existence of the complete event sequence while limiting the amount of information revealed to the other party.

5.5.1 Policy Analysis

Violations of policies are rare events. Because of this, monitoring system collect a large amount of information that does not contribute to violations. Our distributed event-correlation algorithm identifies which events might contribute to violations and shares only those with other security domains.

A violation of a policy is the presence of a particular state on a set of related resources (e.g., in Table 3.1, the resources are a VM instance I, a software P, and a physical machine M). Our system finds violations through the identification of a sequence of events that provide information about changes in the state of such resources. The interaction between monitoring servers in such an example is shown in Figure 5.2.

The structure of the monitoring policy provides the basic intuition supporting our sharing strategy. We see that each event provides information about a set of resources. We can take advantage of the relation between resources to simplify the detection process: we split the problem into a set of steps. At each step, we find events related to a single resource that are potentially involved in the violation. Using the relation between resources, we integrate

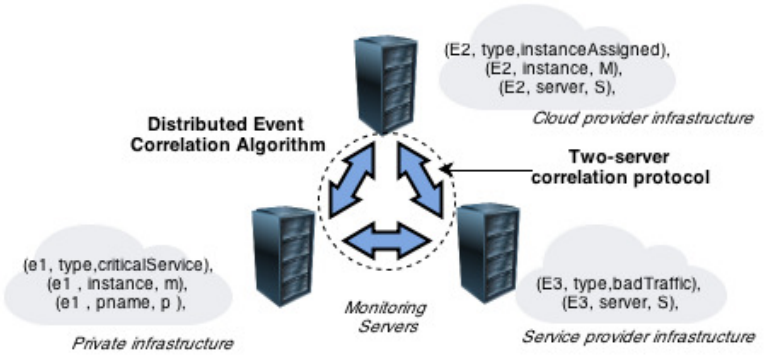

Figure 5.2: Architecture of our monitoring system. Multiple monitoring servers are placed in different security domains. Servers communicate to detect violations of policies.

such partial sequences with events about other resources to reconstruct the complete sequence of events in the violation.

In our case, we consider three resources I, S, and P, where the VM instance I is assigned to the server S, and I runs the program P. In the first step, we integrate the events `badTraffic` and `instanceAssigned` related to the physical server S on the cloud provider system. In the second step, we consider the VM instance I associated to the physical server, and we integrate the remaining events `criticalService` generated by the private infrastructure.

Next section generalizes such an intuition. We use a policy-rewrite process to determine the steps in the event correlation, and we introduce a distributed protocol for identifying with monitoring servers are involved in the detection.

5.5.2 Rule Rewrite

Monitoring servers identify violations by analyzing events related to the resources they manage and by connecting them with events stored in other monitoring servers. We make the relation between resources and events explicit by rewriting the original policy into an equivalent set of rules called *resource-based rules*.

A resource-based rule identifies a sequence of two events potentially contributing to a violation. Our algorithm constructs resource-based rules ensuring that both events in the rule have a resource in common. Because of such a common resource, once a server finds an event matching a portion of the rule, all other events satisfying the remaining part of the rule need

to provide information about the common resource. We use a distributed naming system to identify the monitoring servers storing events about each resource.

As resources involved in a policy violation are related to each other, some events contributing to a violation carry information about two or more of the resources involved. Our system uses these events to connect resource-based rules together and reconstruct the complete sequence of events that may lead to a violation[1]. Fig. 5.3 describes a greedy version of the algorithm we use in the policy rewrite. Intuitively, the algorithm takes a policy P and selects two events with a resource in common (line 2). The two events are substituted with a new logic statement $partial_i$ containing all the information from the previous events (line 4), and a new resource-based rule PR is created (line 5) by taking the two events and the time constraints *timec* involving both of them. If the resulting policy P is composed of two events, the algorithm is complete (line 6), otherwise the execution continues recursively (line 13). If the remaining policy does not have any common variable, a new one is created (lines 8-10). The correctness of the rewrite is shown in Lemma 6

Lemma 6. *We take a set of rules R generated through the application of Algorithm 5.3 to a policy P. We have that a set of events $e_1, \ldots, e_n$ creates a violation of P iif it creates a violation of the set of rules R.*

Proof Sketch. The rewrite of the formula P creates a tree structure, where each node is a rule, and two nodes are connected if the consequence of a rule is a condition in the other node. We prove the correctness using induction on the height of the tree. If the height of the tree is 1, then the condition is trivially satisfied as we have only one rule and $r = P$. Assuming that the height of the tree is n, we prove that the condition is satisfied for $n + 1$. In the $n + 1$ tree, we have an additional rewrite that substituted two events e_1, e_2 and their conditions with a $partial_j$ statement, and created a new rule r_j. The rule r_j contains only a subset of the constraints, and because the set of events satisfies the rules for height n, as the rule r_j is satisfied by the same set of events. Because the statement *partial* maintains all the information about matched events, all conditions that were not taken and placed in r_j

[1]Such a semantic condition is common among monitoring policies: if no relation exists between events, any occurrence of certain unrelated events could create a violation. However, our algorithm handles the case of disconnected events.

```
1: function SubPolicy(V, P, PR)
2:     for all e_1, e_2 ∈ P sharing variable V do
3:         VL = vars(e_1) ∪ vars(e_2) // we select all variables used in the two events
4:         P = (P \ e_1 \ e_2) ∪ partial_i(VL)
5:         PR = e_1 ∧ e_2 ∧ [timec(e_1, e_2)] → partial_i(VL);
6:         if size of P is 2 then return P
7:         else
8:             if P has no shared variable then
9:                 take two events e_k, e_n ∈ P and generate a new random resource r
10:                add a property to e_k and e_n to connect them to r
11:            end if
12:            V' = choose a shared variables
13:            P'= SubPolicy(V', P, PR)
14:            return P'
15:        end if
16:    end for
17: end function
```

Figure 5.3: Policy rewrite algorithm pseudocode.

can still be validated in the original node. Hence, no constraints have been eliminated in this process, and all events that satisfy the rules also satisfy the original policy.

□

As an example, we apply the rewrite algorithm to our example policy of Fig. 3.3. First, we choose the variable S and we consider the two events `instanceAssigned` and `badTraffic` to create the following resource-based rule.

$$\begin{aligned}
&1 : (E_2, \text{type}, \text{instanceAssigned}), (E_2, \text{instance}, I), (E_2, \text{server}, S),\\
&2 : (E_2, \text{startTime}, E_2sT), (E_2, \text{endTime}, E_2eT),\\
&3 : (E_3, \text{type}, \texttt{badTraffic}), (E_3, \text{server}, S),\\
&4 : (E_3, \text{startTime}, E_3sT), (E_3, \text{endTime}, E_3eT),\\
&5 : ([E_2 \text{overlaps} E_3] \vee [E_2 \text{during} E_3])\\
&6 :\rightarrow partial_1(I, S, E_2sT, E_2eT, E_3sT, E_3eT)
\end{aligned} \tag{5.2}$$

When two events matching all the given conditions are found, the statement $partial_1$ is added to the knowledge base to indicate the detection of

a partial sequence of events. The variables $E_i sT$ and $E_i eT$ represent the start time and end time of the events. We rewrite the initial policy using the $partial_1$ statement. As the new statement contains all information from the selected events, all temporal constraints and event conditions can still be applied. The result is as follows.

$$\begin{aligned}
&1 : partial_1(I, S, E_2 sT, E_2 eT, E_3 sT, E_3 eT),\\
&2 : (E_1, \text{type}, \texttt{criticalService}), (E_1, \text{instance}, I), (E_1, \text{pname}, P),\\
&3 : (E_1, \text{startTime}, sE_1 sT), (E_1, \text{endTime}, sE_1 eT),\\
&4 : [E_1 \text{during} E_2] \rightarrow (v_1, \text{violation}, I)
\end{aligned} \tag{5.3}$$

Each resource-based rule represents a step in the correlation process by finding a partial sequence of events. The step is performed within a server or through the interaction of two monitoring servers. Next section shows how our system uses such rules.

5.5.3 Event correlation

Our distributed correlation algorithm splits the process of building subsequences of events potentially violating the policy across multiple servers. Each monitoring server *registers* for managing a set of resources in a distributed naming service, and constructs the sequences of events related to them. Servers receive events about the resources for which they are registered.

The system distributes the load while respecting organization boundaries by using different strategies for distributing resources to monitoring servers. For example, each organization unit can decide to allocate all its resources to a monitoring server; or resources for each organization can be distributed randomly across a cluster of monitoring servers within the organization. We use a distributed naming registry based on Zookeeper [71] for maintaining an assignment between resources and monitoring servers with each organization, and we expose such an assignment to external organizations through a DNS-based interface: a server obtains the monitoring servers managing a resource through the resolution of a name containing a short hash of the resource.

Each couple of events satisfying the constraint of a resource-based rule increases the length of a possible violation sequence. A resource-based rule

connects two events having a common resource in one of their properties. Hence, once a server receives an event matching a part of the rule, other relevant events are found by checking which servers registered for the common resource. For each identified server, our system checks the presence of an event satisfying the rest of the rule using a *privacy-preserving matching protocol*. If a match is found, the event is sent to the remote server. The new event triggers resource-based rules on the remote servers and ensures that the correlation process continues.

However, as monitoring is distributed, events about a resource might be stored in multiple systems that are not "registered" for the resource. Our algorithm uses a "subscription" process to keep track of such servers. When a server s receives an event relevant to r that matches a rule, it contacts the registered server for r to search for matching events. Even when matching events are not found, the registered server maintains a reference to s, as it is known now that such a server contains events relevant to r. When new events are received, the registered servers contacts s for correlation. Additionally, when another server s' requests a correlation for r, the address of s is shared so that s' can correlate its events with s directly. The proof below describes how the reference is used in our protocol.

Theorem 5. *If events $e_1, \ldots, e_n$ satisfies all conditions of a policy, the distributed protocol identifies the presence of a violation.*

Proof. We assume that there exists a sequence of common resources that connects all events, i.e., $r_1, \ldots, r_n$ such that $\forall e_i \exists r_k, e_j : r_k \in e_i, r_k \in e_j$. Lemma 6 shows the equivalence between the resource-based rules and the policy. Hence, it is sufficient to show that our algorithm identifies the existence of the two events matching each resource-based rule. By construction, such events have a resource r in common. Hence, given an event e, we need to ensure that a server finds all events e' that share the same resource r. We have three cases.

1. Both events e and e' are received by servers registered for the resource r. According to our algorithm, when an event is received, the server interacts with all servers registered for such a resource. As both servers are registered, the last event received would interact with the server storing the first event.

2. Event e received by a server s registered for r, and event e' received by a non-registered server s'. If the server receives the events in the sequence (e, e'), the arrival of e' triggers a look up on the naming registry that leads to the identification of the server containing e. If the sequence is (e', e), the server of e is identified, however correlation protocol returns false as e is not present yet, and the event e' is not shared. In this case, s saves the reference for s'. When the event e is received, s runs the correlation protocol with s' again and identifies the event e'.

3. Both events e and e' are received by two non-registered servers s and s'. In such a case, the first event triggers a lookup in the naming system that leads to the identification of a server s_r registered for the resource r. The correlation process creates the reference to s in the server. The reception of the event e' triggers the same process. This time, s_r saves the reference to s' and returns the reference to s. The correlation process between s and s' identifies the matching events.

□

5.5.4 Privacy-preserving Matching Protocol

The privacy-preserving matching protocol is initiated between two monitoring servers: one peer, called the *gc-client*, initiate the process by picking a resource-based rule and an event e for which it wants to find a match. The other peer, called the *gc-server*, considers all local events satisfying the local condition of the rule and, for each event e', it executes a *two-event matching* protocol. To speed up the process, the system executes the two-event matching sessions for all couples (e, e') in parallel.

We use garbled circuits [6] to implement the two-event matching protocol. Garbled circuits are a cryptographic mechanism for performing secure two-party computation. Without the use of cryptography, one server needs to acquire data about both events to validate all constraints (temporal and others) specified in the rule. However, such an approach would reveal a large amount of information to the other party, as all relevant events need to be stored on one server. Using secure two-party computation, each party provides part of the input data and collaborates with the other party through a distributed protocol to determine if two events satisfy the constraints of the

1 : $partial_1(I, E_2sT, E_2eT)$,
2 : $(E_1, \text{instance}, I)$,
5 : $[E_1 \text{during} E_2]$

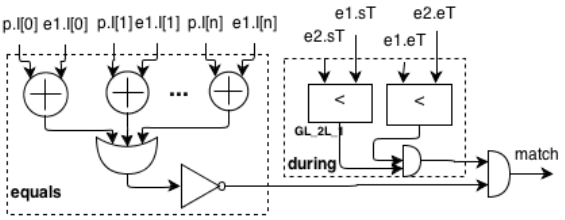

Figure 5.4: (left) Simplified resource-based rule containing only constraints requiring input from both events; the partial statement is simplified to include only variables used in the condition; (right) circuit blocks implementing the resource-based rule.

rule. The data provided by each party remains hidden, while both parties only know the result of the computation. In the last several years, garbled circuits have been shown to be one of the most efficient methods for performing secure two-party computation [72].

Garbled circuit protocols require encoding the computation to perform as a binary circuit. Our system encodes events into binary strings and generates a combinatorial circuit based on the conditions of each resource-based rule. Circuits are created by connecting using AND, OR, or NOT gates subcircuit blocks that depend on the type of constraints specified in the policy. We consider only constraints that require input from both events, as other constraints are validated locally.

The sub-circuit blocks in our implementation cover all temporal constraints in Fig. 3.2, equality, and less-than. More circuits can be created for other types of constraints. For the resource-based rule in Eq. 5.3, the transformation maintains the equality constraint between the values of the variable I used in the event and the statement, and the *during* temporal constraint, as shown in Fig. 5.4(left). Fig. 5.4(right) shows the encoding of the policy.

Our system uses a recent implementation of the garbled circuit protocol [72] to execute the circuit. The gc-client sends the id of the rule to check, and interacts with the server to construct the garbled circuit. The gc-server uses an Oblivious Transfer (OT) protocol [73] to ask the gc-client to encrypt its input, and uses such data to execute the encrypted circuit locally. The encrypted output of the circuit is sent to the client for decryption. If a match is found, the gc-client sends the matched event unencrypted. The gc-server adds the event in its local storage, which might triggers other two-server event correlation protocols.

The system executes the privacy-preserving matching protocol for each couple of events. As each couple is checked independently from the others, parallelism increases significantly the throughput of the system. The garbled circuit computation requires several communication round trip for the exchange of data, so the parallelization allows a better utilization of the CPU which would be otherwise idle waiting to receive events.

In our interactions, the gc-server returns to the gc-client the number of events satisfying the local conditions to determine the number of times the privacy-preserving matching protocol is executed. If such a number reveals information about the state of the infrastructure, the monitoring server can report any number larger than the given value so that the number of events cannot be used to make any inference. Once the local events are exhausted, additional computations are performed with invalid values to ensure that no matching is possible.

5.5.5 Privacy Consideration and Limitations

From a privacy-perspective, the security property of the 2-event matching protocol shows that, for two-event policies, we only share events that create violations. Such a metric is the minimum level of information sharing that we can have between two organizations [74]. However, when the complexity of the policy increases and multiple resource-based rules are needed, sequences of events matching a single resource-based rule need to be shared to process the next resource-based rule. In such a situation, we limit the information to share by selecting, when possible, resource-based rules that are matched rarely.

The interaction between monitoring servers leaks additional information that can be used to make inference on the state of the remote party, even if no explicit sharing occurs. The request for a two-party correlation reveals the hash of the common resource and the policy involved. The hash is kept intentionally short, so that conflicts and false positives are possible, making the identification of the resource id hard. The presence of an interaction, even if leads to no matched events, can still reveal that a subsequence of events matching a portion of the policy is present on the server. To counter such an inference, we add spurious requests with random events to ensure

that such knowledge cannot be inferred.

The implementation of secure two-party computation used in our system relies on the assumption of honest-but-curious attacker [6]. Such attack model assumes that the two parties interact according to the protocol, and do not provide false information about their own systems. In the interaction between organizations, additional mechanisms can ensure that false information is not provided. The periodic auditing currently performed for ensuring compliance to regulations could also validate recoded logs of the interactions. Such logs create an audit trail that could dissuade organizations from providing false information. In addition, techniques have been proposed to validate the received information through independent information sources [75] to ensure its correctness. Moreover, progress has been made for building secure two-party computation that apply to semi-honest adversary [76]. Such advancements can be integrated in our solution.

CHAPTER 6

EXPERIMENTAL EVALUATION

In this chapter, we evaluate our approaches for information sharing to measure the reduction in the amount of information shared between organizations. First, we evaluate our locality-based approach in Section 6.1. Then, we evaluate our resource-based approach in Section 6.2. In both cases, we can significantly reduce the information shared with other organizations. With the use of our resource-based approach, we reduce information sharing to the minimum need-to-know for simple policies composed of two events.

6.1 Locality in Multi-Organization Systems

We perform a set of experiments to measure the ability of our information sharing strategies to reduce the data sent between the two organizations. We quantify the data by measuring the amount of basic information units shared. An information unit is a piece of information that associates a resource with a predicate. For example, an event $\texttt{computer}(host_1)$ associates the resource $host_1$ with the predicate `computer`. We count this event as one information unit. A predicate $\texttt{hasIP}(host_1, ip_1)$ relates the resources $host_1$ and ip_1 to `hasIP` and we consider it composed of two information units. This measure is representative of the information shared and does not depend on the specific definition of the event parameters.

We consider an event dataset collected by monitoring the SNMP state of 10 research hosts for 30 days. These hosts include a mix of laptops, development machines, and a web server. We consider 14 types of messages providing information about resources in the system. The dataset includes information about 500 distinct running programs (associated to 20000 PIDs), 70000 distinct network connections, 50 distinct network services, and 4100 IP addresses. We scale the dataset to represent a larger set of machines by

constructing probabilistic models of the events. The sequence of events in each organization is created by generating events which parameter values are taken from such distributions. Events are timestamped and added to each knowledge base.

As the type of policies that can be specified in a real system can vary widely depending on the interests of the administrators, we evaluate the performance of our approach using a wide range of policies. We randomly generate valid policies which correlate types of events that are semantically related to each other by sharing the values of some variables. We ensure that the shared values are semantically meaningful (e.g., the value for an IP in an event is correlated to the value of IP on another event) by constructing a graph where resource types are nodes, and event types are edges. Starting from a resource type node, we randomly select an event and add it to the policy. We continue until we reach a predefined length of the policy. If we reencounter the same resource type node multiple times, we randomly decide if the event needs to refer to the previous resource (i.e., use the same variable name), or if we expect the event to refer to a different resource (i.e., different variable names).

We create a completeness KB that is consistent with the information collected by our SNMP-based monitoring system. This completeness KB is shown in Table 6.1. The KB of each organization is populated by generating events according to the dataset distribution. We track each resource used in the events and assign it to either D_1 or D_2. The completeness KB allows us to distribute events so that each KB contains the events that would be collected by a monitoring system described by such a completeness KB.

First, we show that our solution limits the overall exchange of information. Without the use of the completeness KB, organization A acquires all events in organization B which are relevant to any of the predicates in the rule. The first experiment measures the fraction of information shared with the increase of the length of the rule. The fraction of information shared is measured as the ratio of information shared over the information relevant to the rule (i.e., events which names appear in the rule). Its results are shown in Fig. 6.1. When we increase the length of the rule, the fraction of information shared remains almost constant in all cases. For small rule lengths, most of the information is found locally in organization A. Using the completeness KB and a pull strategy, we can reduce the number of events that need to be

Domain	**Event**
$M \in D_A$	$\texttt{TCPService}(M, S)$
$M \in D_A$	$\texttt{port}(S, P) \leftarrow \texttt{TCPService}(M, S)$
$M \in D_A$	$\texttt{UDPService}(M, S)$
$M \in D_A$	$\texttt{port}(S, P) \leftarrow \texttt{UDPService}(M, S)$
$M \in D_A$	$\texttt{TCPConn}(M, C)$
$M \in D_A$	$\texttt{LocalPort}(C, PORT) \leftarrow \texttt{hasTCPConn}(M, C)$
$M \in D_A$	$\texttt{RemotePort}(C, PORT) \leftarrow \texttt{hasTCPConn}(M, C)$
$M \in D_A$	$\texttt{LocalIP}(C, IP) \leftarrow \texttt{hasTCPConn}(M, C)$
$M \in D_A$	$\texttt{RemoteIP}(C, IP) \leftarrow \texttt{hasTCPConn}(M, C)$
$M \in D_A$	$\texttt{software}(C, SW) \leftarrow \texttt{hasTCPConn}(M, C)$
$M \in D_A$	$\texttt{connState}(C, ST) \leftarrow \texttt{hasTCPConn}(M, C)$
...	...

Table 6.1: Portion of the CKB of our SNMP-backed monitoring system.

transferred by not requiring information about the local portion of the rule. In our SNMP case, this approach approximately halves the number of events shared by organization B. The push-pull strategy further reduces the amount of information to share. In the SNMP case, we reduce the information sharing to about 20% of the information shared in a full sharing strategy (i.e., no completeness KB). To obtain this reduction, organization A needs to share to organization B about the same amount of information. The optimal amount of information shared from B to A can be obtained by transferring all data from organization A. We see that the minimal number of complete events that needs to be shared to identify all violations is about 10% of the relevant information. If we share only partial events (i.e., single information units), we can further reduce this amount to about 1% of the relevant information. In this case, it is sufficient for organization B to share enough information to pinpoint which resource in A is in violation, without revealing any of its own events that contribute to the actual violation.

The second experiment measures the fraction of information shared with the increase in the amount of events considered in the organizations. We consider a rule of length 5, and we see that the fraction of information that needs to be transferred remains constant and consistent with the previous

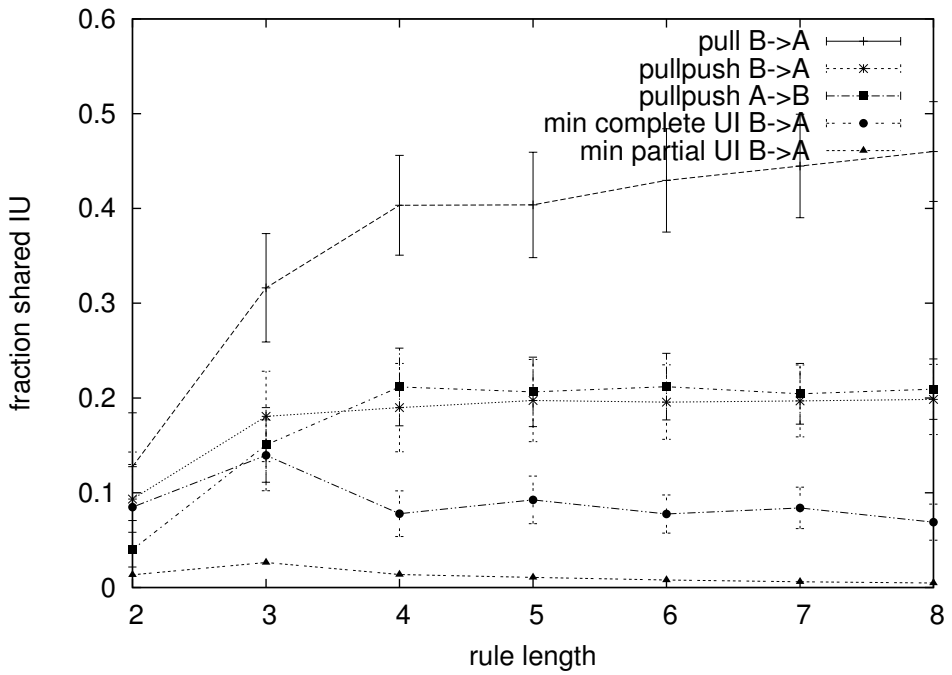

Figure 6.1: Information shared with the increase in the length of the rule.

set of experiments. This data is shown in Fig. 6.2.

Next, we evaluate the overhead introduced by running our information-sharing algorithms. We measure the average amount of persistent queries that are placed on the organization B event stream for selecting the events to share with organization A. For the case of the pull strategy, the number of queries is proportional only to the length of portion of the rule that cannot be evaluated locally. For the case of the push-pull strategy, the number of queries depends also on the size of the events in organization A that we consider. In the push-pull case, the amount of queries is limited to a few hundred. This data is shown in Fig. 6.3.

In summary, our experiments show that our techniques can significantly reduce the number of events to be transferred across the two organizations without significantly increasing the overall load on the system.

6.2 Resource-based Approaches

Our evaluation measured the reduction in the amount of information exchanged when our event-correlation method was used, and the event rate

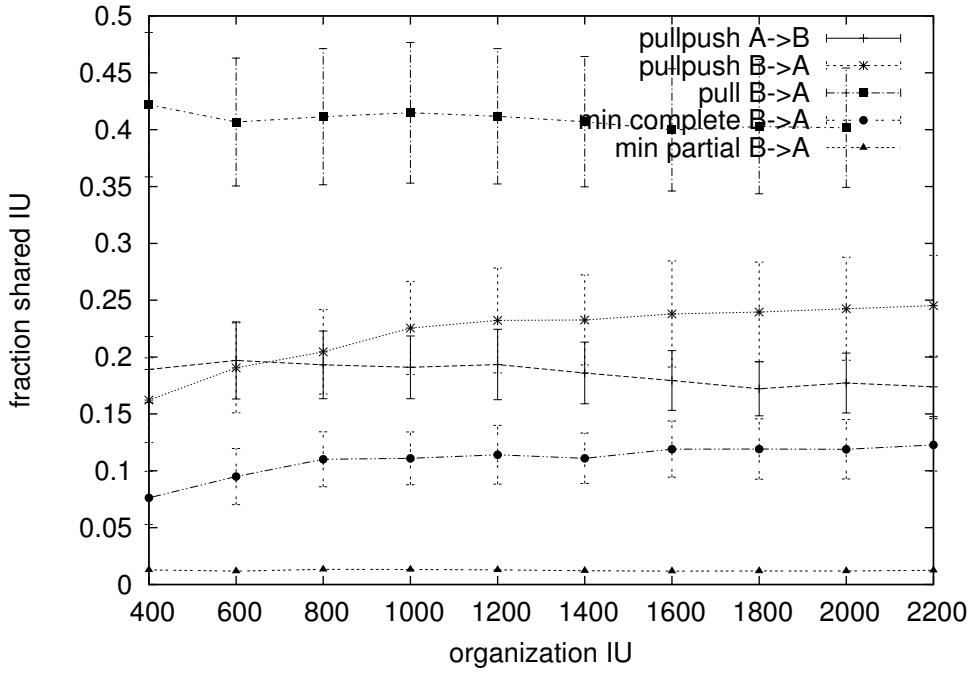

Figure 6.2: Information shared with the increase in the number of events generated by the two organizations. We fix the rule size to 5.

obtained with our two-event matching protocol. We implemented the system in Java and used a garbled circuit protocol implementation by Huang et al. [72], with modifications performed to improve significantly parallelism. We ran our experiments on Amazon EC2 m1.large instances (7.5 Gb memory, 4 compute units), an instance type which computation capabilities fall in the middle of the EC2 spectrum.

6.2.1 Reduction of Event Sharing

We measured the reduction in the information shared between domains. Resources were partitioned across servers, and events were distributed randomly. Information sharing occurs when events about the same resource are stored in different domains. As the occurrence of such a condition is policy- and event- dependent, we evaluated our solution in different points in the space by changing three critical parameters. The first one is the frequency at which two events in different domains create a partial policy violation. The second parameter is the fraction of the infrastructure under the control of each organization. The third parameter is the number of security domains

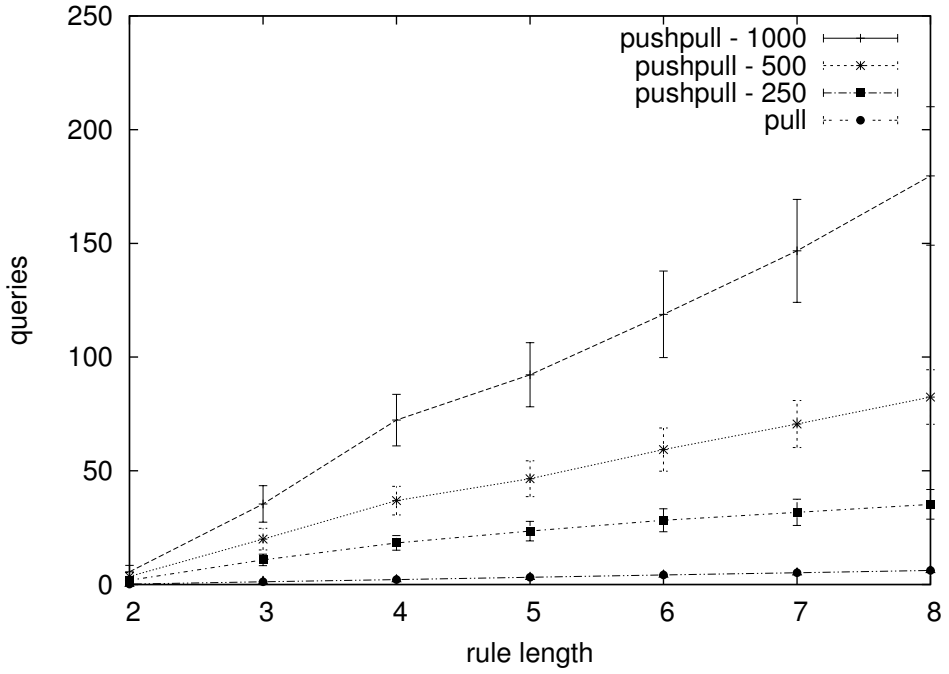

Figure 6.3: Number of persistent queries placed on organization B.

managing the infrastructure.

We measured the performance of our encrypted communication (`encr`) with rules of different complexity (`2-event rule, 4-event rule`). We compared it with a clear-text solution (`clear-txt`) that sends events related to a resource to the monitoring servers managing it (even if they are in a different domain) [77], and with the minimum need-to-know information (`min`).

First, we analyzed how the frequency with which events create partial policy violations affects the amount of information shared. We created events so that each pair of events in a policy has a given probability of referring to the same resource, and we randomly distributed them across domains. We show the results in Fig. 6.4. Our system significantly outperformed the baseline (i.e., `clear-txt`) solution and, for two-event polices, the fraction remained equal to the theoretical minimum (`min`). As we measure events shared over total events, the theoretical minimum number of events for the 4-event rules is smaller than the one for the 2-event rule: in the optimal case, a single interaction can summarize information about multiple events and it is counted as one event.

The distribution of resources across domains affects the fraction of events shared, as shown in Fig. 6.5. To test the system under less than ideal condi-

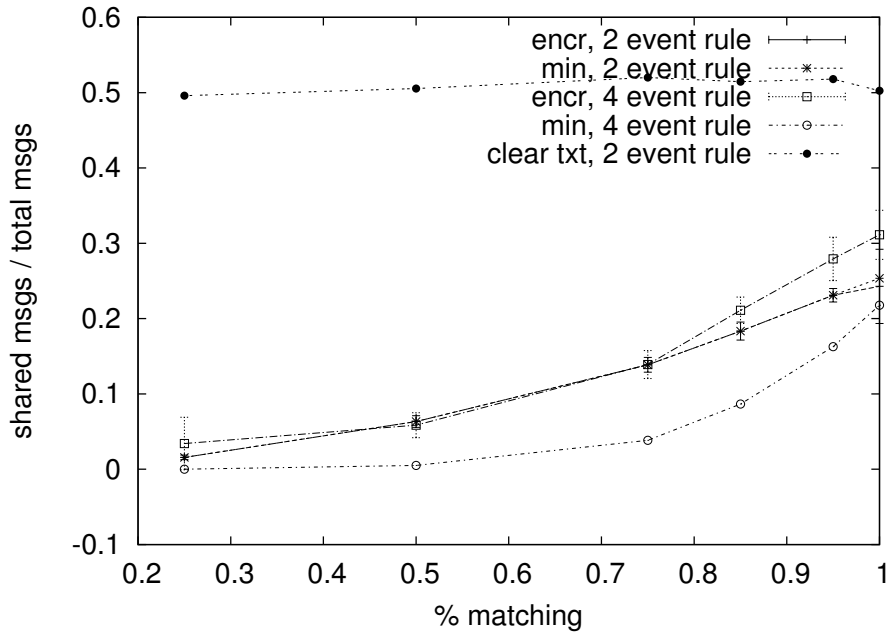

Figure 6.4: Probability of matching events affects the fraction of events shared.

tions, we created events that partially matched policies with a probability of 75% and we distributed them randomly to each server. The highest information sharing occurred when each organization had half of the resources, while the amount of event shared is reduced when more resources are managed by a single domain.

We measured the fraction of events stored at each server under different conditions (see Fig. 6.6). We considered both complete events and events that can be inferred from the presence of partial statements. We saw that increasing the number of security domains reduced the average number of events stored in each server, and that, in all cases, our system provided a significant improvement compared to a clear-text solution.

To summarize, our experiment showed that while the performance of the system depends on the conditions of the policy and the frequency of matching events, our solution still outperforms a baseline solution. In many cases, the amount of information shared is close to the minimum possible. Best conditions occur when events in different domains creating a policy violations are not frequent, and when a significant fraction of interacting resources are stored within the same security domains, so that most violations can be found locally.

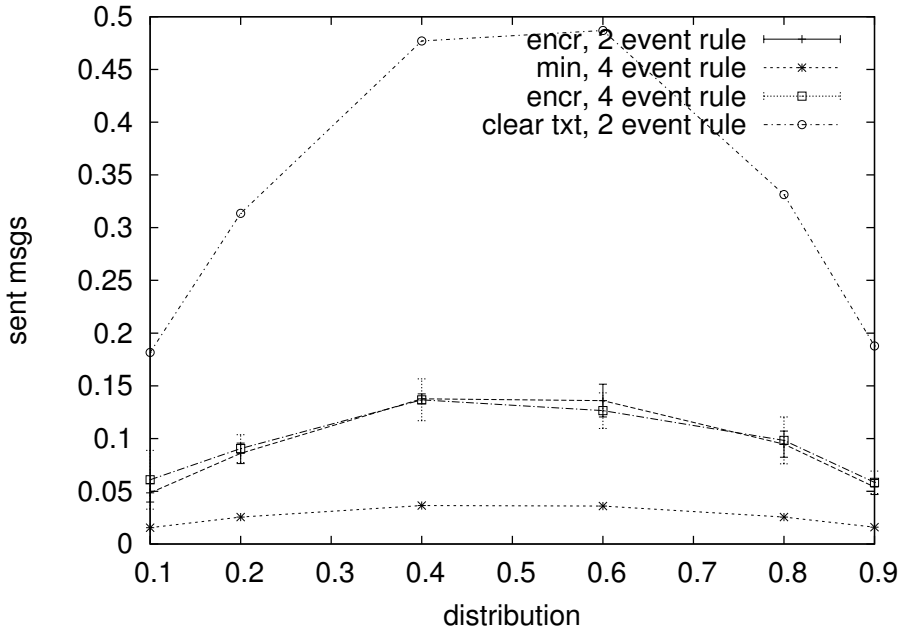

Figure 6.5: Fraction of resources allocated to a monitoring server. 2 servers

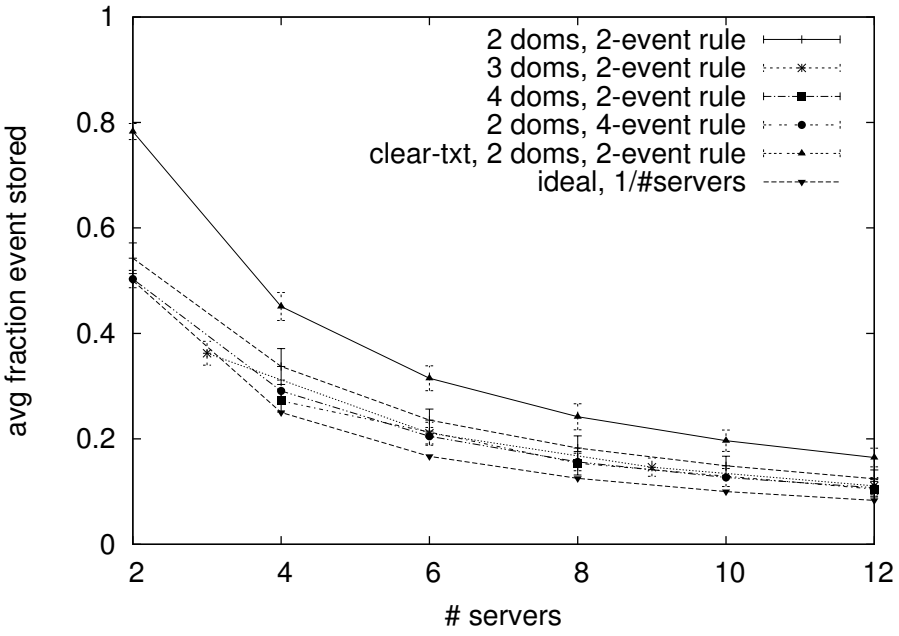

Figure 6.6: Average fraction of events known to each server.

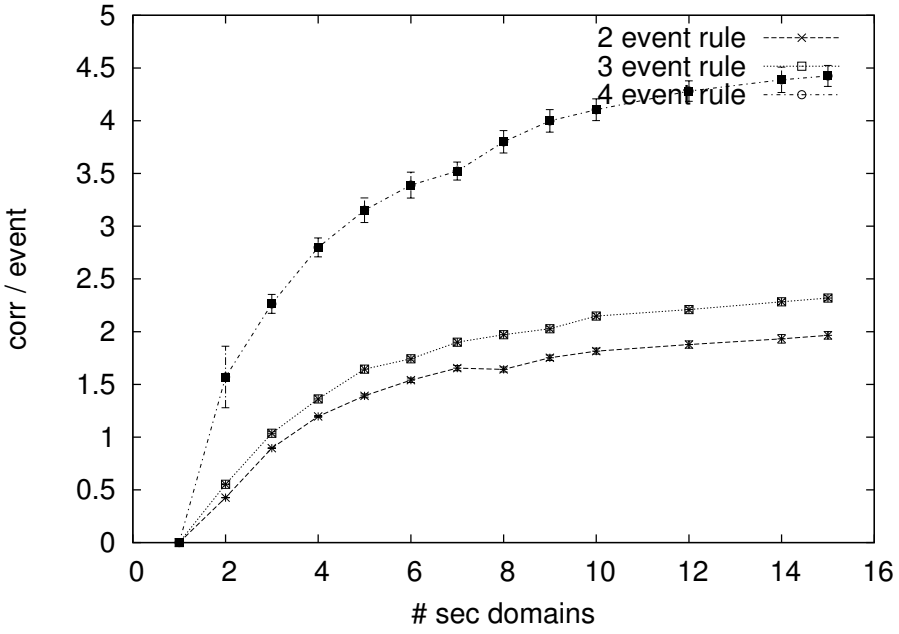

Figure 6.7: Server load, multiple security domains. One server per domain.

6.2.2 Performance Evaluation

To evaluate computation overhead, we measured the average number of secure two-party computations performed by each server, as shown in Fig. 6.7. We varied the number of domains. We considered the ratio between two-event correlations and events received. We saw that increasing the number of security domains increases the number of two-event correlations performed, as it increases the likelihood that interacting resources are managed by external servers.

We measured the ability of parallelizing and of distributing the computation by measuring the average number of computation per-server, as shown in Fig. 6.8. We injected a constant number of events into the system and we measured the ratio between the average number of two-event correlations performed on each server and the total number of events. When we increase the number of monitoring servers within each domain, the average number of per-server two-event correlations decreases as resources are distributed across the multiple servers.

To demonstrate the practical feasibility of our system, we measured the event rate achievable using our prototype implementation on an Amazon EC2 deployment. We used 64 bits for representing the resource name in bi-

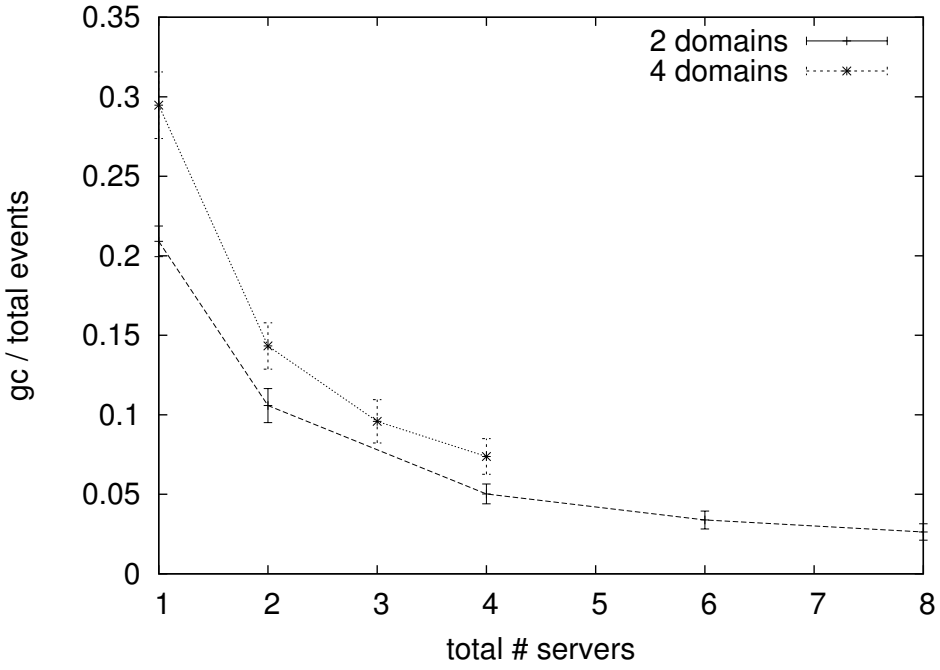

Figure 6.8: Distribution of load with the increase of the number of servers within each domain.

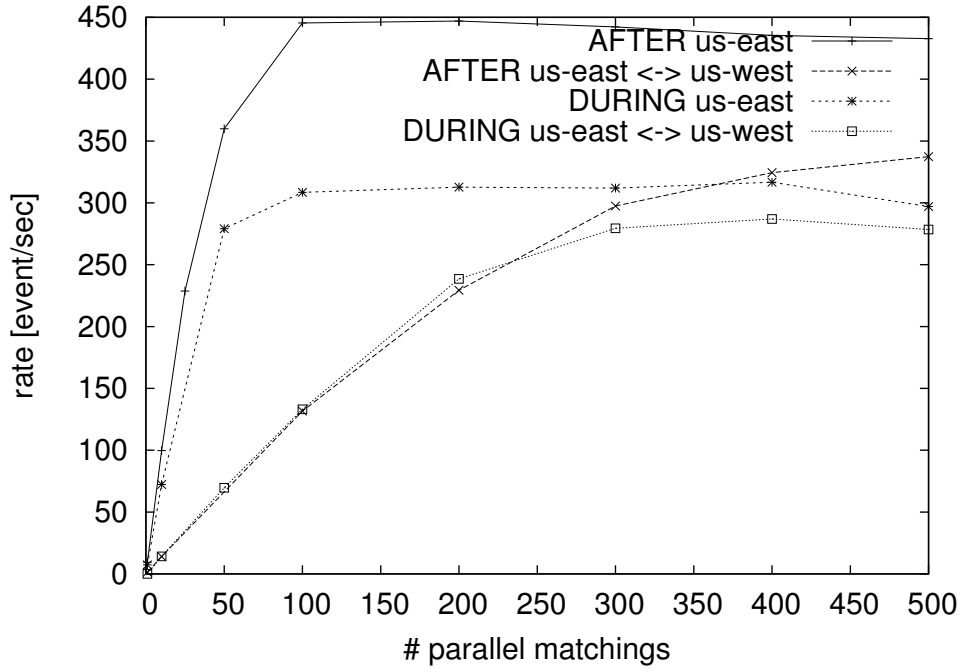

Figure 6.9: Delay in the processing of an event as a function of the level of concurrency in the server.

nary form, and 32 bits for representing each event timestamp. Such values are sufficient to reduce collisions and to maintain low circuit complexity. Because the performance of garbled circuit protocols depends on the round-trip communication delays, we measured the performance between two servers within the same geographical region, and between servers in different regions. The former represents conditions found when monitoring servers are co-located within the same provider, while the latter when servers are at different providers.

We measured the throughput in event correlation per second and we show the results in Fig. 6.9. The remote dataset was split into groups of 100 events, and the processing of each group occurred in parallel. When using multiple threads, we increased the rate up to 400 correlations per second on a single server.

We evaluated the effects of the policy constraints on the system's throughput. We evaluated two circuits: one checking for an equivalence between properties and for a constraint *after*; and a more complex circuit checking for an equivalence and a constraint *during*. The first one had an input size of 192 bits: 64 bits for each property name and 64 bits for two timestamps. The second circuit used 256 bits: 64 bits for each property name, and 128 bits for the four timestamps. The complex circuit reduced the throughput by about 30%.

We evaluated the effect of colocation of servers on event throughput. We considered servers co-located in the us-east region, and servers placed in us-east and us-west. Without concurrency, co-located servers for the *after* constraints obtained a rate of 9.5 event/second. When servers are located in different regions, the rate went down to 1.4 event/second. However, increasing the concurrency significantly increased the event rate. With 500 concurrent executions, co-located servers obtained a rate of 435 event/second, while servers in different regions obtained 337 event/second, with a performance reduction by about 22%. The *before* constraint had similar results.

In summary, our experiments demonstrated that our system can obtain a performance of hundreds of correlation per seconds on each server, and we can distribute the process across multiple servers.

CHAPTER 7

EVALUATION OF THE SECURITY OF THE ARCHITECTURE

Monitoring systems are at the core of the security of organizations. Malicious manipulation and access to the information collected by a monitoring system can create significant challenges in the management of systems. In the previous chapters we focused on an honest-but-curious attack model. In such an attack model, organizations interact by behaving according to the protocols while attempting to learn information amount the other parties during the process. Such a collaboration is realistic in several situations. First, when organizations have an advantage in collaborating, for example for protecting against attacks or for detecting other problems that would affect their infrastructure. In such a situation, deviating from the protocol would inhibit the ability of the overall system to operate correctly. Second, the honest-but curious model is appropriate when regulatory agencies mandate collaboration to rules, and implement periodic auditing to ensure that systems are behaving according to the protocols.

In this chapter analyses the effect of relaxing our assumption about honest-but-curious attackers by analyzing situations where monitoring systems within each organization are compromised. While our architecture cannot provide a complete protection against such attacks, we show that, in several situations, our architecture is able to limit their impact to only a subset of the system. A single compromise cannot compromise the security of the entire architecture.

The compromise of servers in the monitoring system causes two problems: it reduces situational awareness by injecting false data, and it provides attackers important information useful for additional attacks. We analyze security by determining the effects on the monitoring architecture of attacks on integrity, confidentiality, and availability of the data. The first three sections focus on integrity and on availability. Section 7.1 analyses the possible goals of attacks directed toward the integrity and the availability of the monitor-

ing data. Section 7.2 defines attack models targeting integrity. Section 7.3 uses such models to analyze the security of our system. Additionally, the section describes countermeasures that we implement in our architecture for defending against such attacks.

The remaining sections in the chapter describe the resilience of our architecture toward attacks against the confidentiality of the data stored in the monitoring system. Section 7.4 defines attack models targeting confidentiality. Section 7.5 analyses the advantages of our architecture in such scenarios. Section 7.6 presents an experimental evaluation of such a protection.

7.1 Integrity and Availability of the Monitoring Information

Compromising the integrity or the availability of the monitoring system can significantly affect the ability of the system to detect threats and to respond to them.

When the availability of the monitoring system is compromised, the system loses the ability of detecting new attacks or problems in the infrastructure. For example, an attacker could affect temporarily the availability of the monitoring system to hide malicious actions that could indicate the compromise of systems. Additionally, an attacker could disable the monitoring system to reduce the ability of responding to attacks in situations such as Denial-of-Service attacks, where fast online response is required.

Compromising the integrity of a system has similar effects, but the problem is intensified by two facts. First, there is an increase difficulty in detecting that the attack against the monitoring system is even occurring. While availability attacks can be detected using failure detectors in distributed systems, integrity attacks require more complex detection mechanisms that are generally harder to implement. Second, an attacker can inject false information in the monitoring system to trigger an inappropriate response. For example, during the response to a DDoS attack, a compromised monitoring system could provide false information that points the operator to the wrong cause of the problem. In such a situation, an operator might stop legitimate network flows believing that they are part of the attack.

Our security analysis focuses on the effects of compromises of parts of the

infrastructure. While an attack to the availability of the infrastructure is easier to succeed than an attack to the integrity, its effects on the monitoring system are a subset of the effects of a compromise of the integrity of the data. If the integrity of a device is compromised, an attacker can decide to make it stop operating. Hence, our analysis focuses on integrity attacks as a worst-case scenario.

7.2 Integrity Threat Scenario

We model an attacker interested in compromising a policy-based monitoring system. We assume that the attacker is interested in affecting the operation of the monitoring system by hiding violations or by injecting false violations. Hiding violations would allow the attacker to operate in the system without detection, while injecting violations would allow the attacker to trigger indirectly specific security responses.

We classify the attacker's goals in two categories, as follows.

1. INT_RESOURCE: An attacker is interested in controlling violations regarding a specific resource.

2. INT_ALL: An attacker is interested in creating or hiding violations without any specific preference for the resources involved.

The INT_RESOURCE attack is a targeted attack directed at modifying specific information about a resource. For example, attackers could modify information about the programs running on a machine for hiding malicious software, or they could modify traffic flows to hide malicious communications. The INT_ALL attacker is interested in disrupting the monitoring system by injecting false information, independently from the resources involved. Attackers with such a goal are interested in maximizing the number of violations that can be hidden or created by attackers.

We provide a quantification of the difficulty of the attack by making a few simplifying assumption. Such quantification is useful for comparing the security of the two solutions. First, we define a model for the attack. We assume that an attacker can compromise a system by taking some effort (e.g., sending fishing emails, installation of malware). Such an effort could be

composed of multiple tries, multiple steps, and can be distributed over time. We assume that there is a probability p_c of such an effort being successful and resulting in the compromise of the target monitoring server. Additionally, we assume that each effort has a probability p_d of being detected. For simplifying formulas, we assume that the probabilities are the same in the centralized solution and in the distributed solution: in both cases, the servers should be secured according to best practices.

7.3 Availability and Integrity of the Monitoring Data

We analyze the effects of compromises on the integrity of the monitoring process, and we propose a set of countermeasures that can alleviate the problem through the use of redundant sensors and redundant servers. We consider two basic types of attacks to the infrastructure. The first type of attack focuses on compromising information sources (e.g., end-hosts, IDS systems) that provide security information to monitoring servers. The second type of attack focuses on compromising the monitoring servers.

7.3.1 Compromise Information Sources

An attacker can compromise directly the information sources providing information about the target resource. For example, an attacker interested in hiding network traffic can compromise the IDS node collecting information about flows. If the attacker can prevent the security information reaching the monitoring servers, all violations that rely on the compromised information are under the control of the attacker.

With this type of attack, INT_RESOURCE attackers can focus their efforts on compromising the information sources monitoring the resources they are targeting. An INT_ALL attacker can maximize the number of information sources that need to be compromised. Increasing the information sources that are compromised increases the violations that can be controlled by the attacker.

7.3.2 Mitigations

We mitigate the effects of attacks to the information sources in two ways: hardening of information sources, and redundant monitoring.

A large amount of work has been developed on hardening information sources so that they are less vulnerable to attacks. Recent approaches use Virtual Machine Introspection (VMI) (e.g., Garfinkel et al. [46]) to separate the monitoring infrastructure from the monitored system, so that attacks to the latter does not affect the former. In our implementation, we use such techniques when monitoring running processes and network connections in a virtual machine environment.

Redundancy in the information sources is another method for protecting the system from compromises. In several situations, information about a resource can be acquired through multiple independent sources. For example, network flow information is observed both by IDS systems and by the hosts that are part of the connection. As each information source creates events independently, the monitoring servers can integrate such information and detect suspicious situations (e.g., an event is received from one information source instead from all redundant information sources).

7.3.3 Compromise of Monitoring Servers

Another attack vector is the compromise of the monitoring servers. By compromising servers, attackers have complete access to all the resource information managed by the server. For example, an attacker can change the processing code and inject false information in the system, or prevent violations from being detected. We estimate the resilience to compromises by evaluating the effects of attacks on availability and integrity of the system.

We consider an architecture based on multiple monitoring servers placed in different security domains. We compare the security of our solution with a centralized solution where all information are integrated on a single common server for analysis.

For a INT_RESOURCE attacker, the different between the two systems is limited. In both cases, obtaining control of the monitoring server controlling a resource requires compromising one single monitoring server. In the centralized solution, such a server is the common server, while in our dis-

tributed solution, such a server is the server within a security domain that is managing such a resource. In our attack model, in the centralized solution, we have that a single successful effort for obtaining access to the machine gives complete control to the monitoring system. Hence, the probability of having the system compromised is p_c, and the probability of detecting such an attack is p_d. The result for the distributed solution is the same. There is a probability p_c that a specific server in one of the security domains is compromised, and a probability p_d that the attack is detected.

For a INT_ALL attacker, the effects of attacks on the two architectures are different. In the centralized architecture, a single compromise would permit users to obtain complete control of all resources in the system. In the distributed architecture, obtaining such a complete control requires compromising all servers in the different security domains. In our attack model, such a complete compromise in a system with n servers has a probability of success of p_c^n, and a probability of detection of $(1 - p_d^n)$ (i.e., at least one security domain detects the attack).

When moving to a distributed solution, we can have partial compromises of the system: some resources can be under the control of an attacker, while others are not compromised. Because the INT_ALL attacker is not interested in compromising a specific server, it can take advantage of the increased attack surface for increasing the success of partial attacks. If we assume that resources are uniformly distributed across servers, controlling a portion r of the resources requires compromising any single server.

Hence, the probability of having exactly r resources under the control of an attacker can be modeled as the success of at least one of the compromise efforts. We express such a probability using Bernulli trials as follows.

$$\binom{n}{1} p_c (1 - p_c)^{n-1} \tag{7.1}$$

As the effort is spread across multiple organizations, the probability of detecting the attack increases. The attack is detected if any of the organizations detects the attack as $(1 - p_c)^n$.

To evaluate the impact of controlling resources on the detection of violations, we consider the resource-based algorithm for the distributed detection of violations. In such an algorithm, the communication between monitoring servers for each violation is organized in a tree-like structure. The monitoring

servers involved in the violation are the nodes of such a tree.

Once an attacker has control of a resource, she can hide real violations or create fictitious ones by injecting false information into the processing flow of the distributed algorithm. For hiding an existing violation, it is sufficient to suppress a message specifying the partial match of the policy. For creating a new violation, an attacker can create a new message indicating the presence of partial matches. If the condition of the other resources involved is correct, then a fictitious violation is detected.

In summary, the impact of attacks targeted to a specific resource remains similar in the centralized and the distributed architecture. However, catastrophic failures where all the monitoring system is under the control of an attacker are harder to obtain. The system can fail partially, while containing to work for the majority of the resources involved.

7.3.4 Mitigations

As in the previous section, we use redundancy to limit the impact of attacks. We implement redundancy in the processing of the algorithm by having multiple monitoring servers manage each resource.

The redundant servers receive, directly from the event sources, information for the resources they monitor. Each server integrates such information with the partial results generated from other servers, and forwards the computed results to the next servers according to the distributed algorithm.

Two main changes are introduced for supporting a redundant computation. First, we use a majority voting for accepting partial computation results coming from other servers: if the same partial result is received from a majority of the redundant servers, the server adds it to the knowledge base. Otherwise, the result is ignored until enough votes are received. Second, the partial results generated by the server are submitted to all monitoring servers registered for the resource.

Through such a use of redundancy, creating or hiding violations requires compromising a majority of the servers at each level. The tradeoff is an increase number of servers used in the processing, and an n-fold increase in the number of messages exchanged in the system.

7.4 Confidentiality of the Monitoring Data

The effect of distributing the computation across multiple systems helps reducing the impact of compromises on the confidentiality of the data contained in the monitoring system. Instead of concentrating information in a single system, we distribute knowledge about the infrastructure into multiple monitoring servers that collaborate for detecting violations of security policies. The centralization of information used in other monitoring systems relies on the assumption that securing a single system is simpler than securing multiple systems. However, recent compromises of critical systems such as certification authorities [78], targeted attacks [79], and the presence of zero-days vulnerabilities challenge such an assumption. For example, the exploitation of a single zero-day vulnerability in the monitoring server would allow an attacker to acquire all information about the infrastructure. In our system, the exploitation of a single monitoring server would reveal only limited information.

7.4.1 Confidentiality Threat Scenario

We define an attacker, Eve, interested in acquiring more information about the state and the structure of the infrastructure system. We assume that the attacker has a limited initial knowledge and she is interested in compromising the monitoring system to acquire more knowledge. While compromising the monitoring system is not the only method for acquiring more information about the system's structure (e.g., she could compromise a set of servers in the network and monitor traffic and communications), we assume that attacking it maximizes the efficiency of the attack. We assume that the attacker has knowledge about the monitoring system.

First, we make a few reasonable assumptions about the capability of the attacker for compromising hosts. We assume that attackers can compromise monitoring servers, but these compromises are rare events. We assume that some effort is required for compromising an additional monitoring server (i.e., an attacker cannot compromise all servers at the same time with no effort). Such an effort can represent the difficulty of acquiring an additional set of credentials, or the difficulty of compromising another machine located in the network of the monitoring server so that firewall protections can be

circumvented.

Then, we define more clearly the goals of the attackers. We assume that the attacker is interested in both the presence of policy violations and in the raw events collected by the monitoring system. Policy violations indicate directly possible venues for attacks, but they are generally fixed quickly by network administrators. This provides a limited window of opportunity to attackers. Other data can be used to identify critical systems or plan the next step of the attack.

The type of raw events interesting to an attacker can change depending on the type of monitoring performed by organizations. We consider multiple types of attackers interested in acquiring different types of information from the monitoring system. We classify attackers as follows:

1. MAX_ALL: An attacker wants to maximize her overall knowledge about the system.

2. MAX_RESOURCE_CRITICAL: An attacker wants to maximize her knowledge about a limited set of critical resources.

3. MAX_TYPE_CRITICAL: An attacker wants to maximize her knowledge about specific critical types of events (e.g., the presence of a vulnerability on machines).

We evaluate the protection provided by our system to these different types of attackers, and we show that our resource-based distribution of events provides a better protection than other solutions.

7.5 Protecting Data from Attacks

Distributing the knowledge about the system to multiple monitoring servers improves the security of the system toward our attack model for several reasons. First, the fact that a single compromise is not sufficient anymore for acquiring the information searched by attackers forces them to perform multiple actions. The increased activity gives multiple opportunities to IDS systems to detect such malicious behavior. To get a qualitative idea of this effect, we use a simple probability model. If we consider the probability of

detecting an attack at each action independent p_a, we have that the probability of detection p_{dt} grows with the number of servers k as $p_{dt} = 1 - (1 - p_a)^k$. With no distribution, this number is constant and equal to p_a.

Second, a centralized monitoring system provides a simple target for attack. While rare, zero-day vulnerabilities or stolen credentials can be used by an attacker for compromising the system and, hence, accessing the entire monitoring information. In our distributed approach, we rely on monitoring servers managed by different organization units. The servers are placed in different networks and they are managed using different credentials. In such a configuration, accessing the entire state of the system requires compromising multiple credentials or exploiting multiple zero-days vulnerabilities to communicate with the different servers. Even when a single monitoring server is compromised, such an exploit has limited effects on the amount of information obtained by the attacker.

We can qualitatively estimate the effects of this advantage. For ease of calculation, we consider a simplified attack model where we assume that an attacker has a probability p of successfully compromising a host. Such a probability is related to a simplified notion of the "effort" required for comprising an additional server: p represents the probability of success given a constant effort. In the centralized case, the probability p_c of compromising the central server is equal to p and represents the probability that the entire knowledge about the system is compromised. In our case, an attacker that wants to have access to the information needs to compromise multiples servers. We call p_d the probability of compromising at least k servers over the n monitoring servers, and we define it as follows:

$$p_d = \sum_{i=k}^{n} \binom{n}{i} p^i (1-p)^{(n-i)} \tag{7.2}$$

Distributing the information across n servers is better than centralizing it if $p_d < p_c$, i.e., if the probability p_d of compromising at least k servers containing the information searched by the attacker is lower than the probability p_c of compromising the centralized server. To maximize the difference between the p_d and p_c we need to have a high value of k: we need to force attackers to compromise multiple machines to find the information they need. Such a number k does not need to be very large: if we have 20 monitoring servers and we consider a probability $p = 0.01$ for an attacker to compromise one of

the monitoring servers, a value of $k = 3$ leads to a probability $p_d = 0.001$. This value is robust to changes in probability p. For example, we can consider the common belief that "securing one server is easier than securing multiple machines." To analyze its effects, we can assume that compromising the centralized monitoring server is twice as hard as compromising one of the monitoring servers. Even in this case, if we take a probability $p_c = 0.01$ and a probability of compromising one of the monitoring servers as $p = 0.02$, the probability p_d with $k = 3$ is 0.007. With $k = 4$, this probability goes down to 0.0006.

Hence, while distributing the information increases the attack surface and might make finding one single vulnerable monitoring server more likely, attackers need to compromise multiple machines to acquire the information relevant for their attacks. The multiple steps of the attack make it easier to detect and less likely to succeed completely. Based on this observation, we build our policy-based monitoring system so that information is distributed uniformly across several monitoring nodes. Our experiments show that we can maintain the value k large for the attack models we consider.

The process of matching events within each monitoring host is performed by a service called "broker." A broker is a software service that can run in a dedicated machine or can be co-located with other services. Each broker receives events related to a limited subset of the resources of the organization. They use our algorithm for exchanging information and, hence, for finding all violations presents in the system. In this way, event load and information about the system are distributed across a large number of hosts. Resources are assigned to brokers at random, so that no single host concentrates the knowledge about the resources of a critical organization unit. Each monitoring server can subscribe to violations of specific rules or to violations involving specific resources. These subscriptions are distributed across brokers. Once a broker finds a violation, it delivers a notification to the subscribed monitoring servers. We limit the amount of subscriptions that each server can submit. While communication between brokers is necessary, such communication is limited only to the event-correlation service. Other services can be isolated through firewalls to reduce the attack surface. An architecture supporting our algorithm is shown in Fig. 3.1.

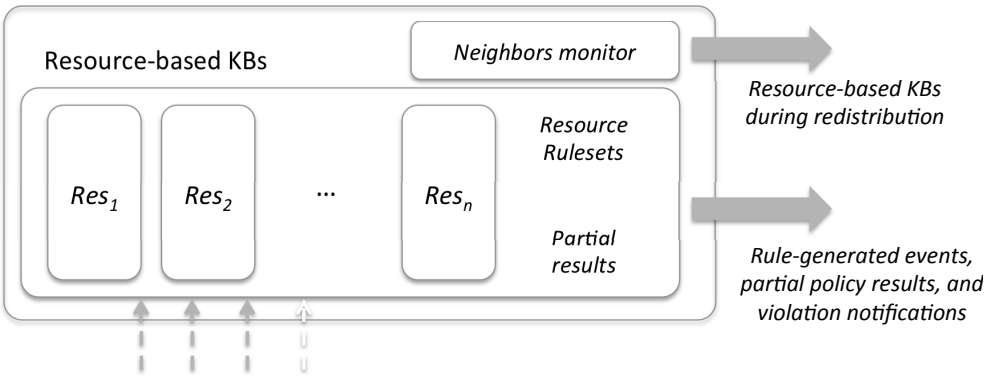

Figure 7.1: Internal structure of our broker implementation. It receives events and adds them to the appropriate resource-based KB. The Datalog resource rulesets are added to the KB obtained by the union of the resource-based KBs. The events generated by the rules are forwarded to the appropriate brokers. In case of a changes in the availability of other brokers, the neighbors monitor sends and receives resource-based KBs from other brokers.

7.6 Experimental Evaluation of the Confidentiality Protection

We implement the policy compliance algorithm in our distributed monitoring system. Our agent-based system provides monitoring of SNMP data, running processes, network connections, and logged users. Information is integrated in a set of monitoring servers communicating using a 1-hop DHT. The hash of the resource name is used for mapping resources to brokers. The system is implemented in Java. Rules are distributed to the servers before starting the process.

Each broker analyses the policies and creates resource rulesets and queries. Events received by brokers are placed in a resource-KB depending on the resources contained in the event. Rulesets and queries are applied on the KB obtained by taking the union of all resource-KBs. Once inference is performed, the monitoring server applies the queries defined from the rules and creates the set of events to send to other brokers. Such events contain the partial executions of the policies or specify the detection of a violation. If a violation is detected, the broker forwards it to the brokers subscribed to receiving violation notifications about the rule or about the resources involved (e.g., the monitoring server in the sub-organization managing the resources).

Mapping between resources and brokers can change over time (e.g., because

we add a new broker). A component, the neighbors' monitor, detects when a resource becomes mapped to another broker and moves the proper part of the knowledge base to such a broker.

Failures of a broker are handled by remapping automatically resources to new brokers. Such a process is managed by the neighbors monitor. However, when a broker fails, its current knowledge base is lost. For the cases in which events are correlated within a limited time window, a failure would only affect temporarily the ability of detection violations until the time window is passed. After such a time, the new broker would have received all relevant events and the event correlation becomes complete again. For the cases in which correlation requires to store longer term information, devices are configured to send periodically such information about the state to brokers. For example, SNMP information is generally long-lived. A device could periodically (e.g., every 10 minutes) send again its entire state to the brokers. A proper handling of timestamps (such as the one described in Walzer et al. [55]) ensures that policy violations that occurred during the downtime are still detected, even if with a delay. An architectural description of a broker is shown in Fig. 7.1.

The evaluation of our system requires having policies and events to correlate. To evaluate our system in a wide range of realistic conditions, we generate events from publicly available data traces and from monitoring a set of systems in our research infrastructure. The first dataset we use is a network trace collected during the three days of the 2010 Network Warfare Competition [80]. We use Snort[1] to analyze the trace and generate 60152 security-relevant events carrying information such as type of alert, source IP, destination IP, protocols, and ports. The second dataset is composed of syslog data collected by monitoring the wireless network infrastructure of Dartmouth college [81]. It is composed of 30 million syslog entries describing the state of the wireless access points. It reports events such as association of wireless devices to access points, interactions between access points, and errors. The third dataset contains SNMP data about configurations of hosts, network connections, running services, and running programs. We collected this dataset by monitoring using SNMP the state of different types of machines: servers, development desktops, and laptop computers. We choose

[1]http://www.snort.org

IDS Resource	Cardinality	With type	Avg Msgs	stddev
Event type	24	24	4.17%	12.0%
Src port	180	357	0.556%	1.97%
Dst port	249	704	0.402%	1.31%
Src IP	75	742	1.33%	4.64%
Dst IP	171	1299	0.585%	1.93%
ICMP type	6	108	16.7%	29.9%
Wireless Resource	**Cardinality**	**With type**	**Avg Msgs**	**stddev**
Event type	42	42	6.67%	17.2%
Access point ID	544	8944	0.184%	0.451%
MAC address	9251	59473	0.010%	0.100%
SNMP Resource	**Cardinality**	**With type**	**Avg Msgs**	**stddev**
Event type	14	14	7.14%	24.8%
Host IP	4145	4147	0.024%	0.457%
Program	493	493	0.20%	1.50%
Network Port	20770	24222	0.00480%	0.20%
Services	56	56	1.79%	4.51%

Table 7.1: Resources and their distributions in the datasets. We consider the average number of messages for each resource and its stddev. The values are normalized by the total number of messages dealing with the specified type of resource.

the datasets to show the applicability of our technique to different data used in policy compliance: network traffic data, network management data, and security management data.

7.6.1 Event Dataset Analysis

We start our evaluation by analyzing the characteristics of the events generated in network management and intrusion detection. Such an analysis shows that a resource-based mapping can provide a more uniform distribution of data than a mapping based on event-type.

First, we analyze our datasets to characterize the distribution of events and of resources. We identify event types and resource types. We have 24 types of events in the IDS dataset, 42 in the wireless management dataset, and 14 in the SNMP dataset. Each dataset has several types of resources. We identify source ip, dest ip, source port, dest port, and ICMP type as different resources for the IDS dataset[2]. We identify access point IDs and MAC addresses for the

[2]We consider source and destination IPs and ports as different resources to better

wireless dataset, and we identify IP, programs, network ports, and services as resources for the SNMP motoring dataset.

We show that the distribution of the number of messages is more flat if we distribute messages by resource, while it is far from uniform when we distribute messages by type (i.e., each resource has a similar number of messages related to them, while there are type of events that are much more frequent than other types). We compute the standard deviation of the normalized distribution of the number of messages by type and by resource. For each event type and resource, we compute its fraction of the total messages. We compute the stddev of this distribution and we represent the value as a percentage. This value is much larger when messages are aggregated by event types, while it remains low (in most cases) when events are aggregated by resource. We summarize this data in Table 7.1.

A graphical representation of the CDF of this message distribution for the Dartmouth dataset is shown in Fig. 7.2. The y-axis of the graph represents the fraction of resources (or message type) that are mentioned in at least the fraction of messages specified in the x-axis. We see that we have a large number of messages for each event type, while most resources are mentioned in a small fraction of the messages. Hence, a distribution of messages by resource could provide a more uniform distribution of information.

7.6.2 Deployment on EC2

We run the system on EC2 spot instances to test the system in a real distributed environment. One of the instances generates events according to the distributions specified in our datasets to model a stream of events from a real system. We performed several experiments by changing the number of policies, the length of the policies, and the number of brokers. Each data point is the average of at least 5 executions. Each time new policies have been generated to consider different variations of rules and events.

We compare our system with systems presenting a different distribution of events. Centralized solutions do not provide any protection against the attack, as all information is compromised as soon as the main server is compromised. For this reason, we do not consider it in our evaluation. Instead,

understand the dataset characteristics.

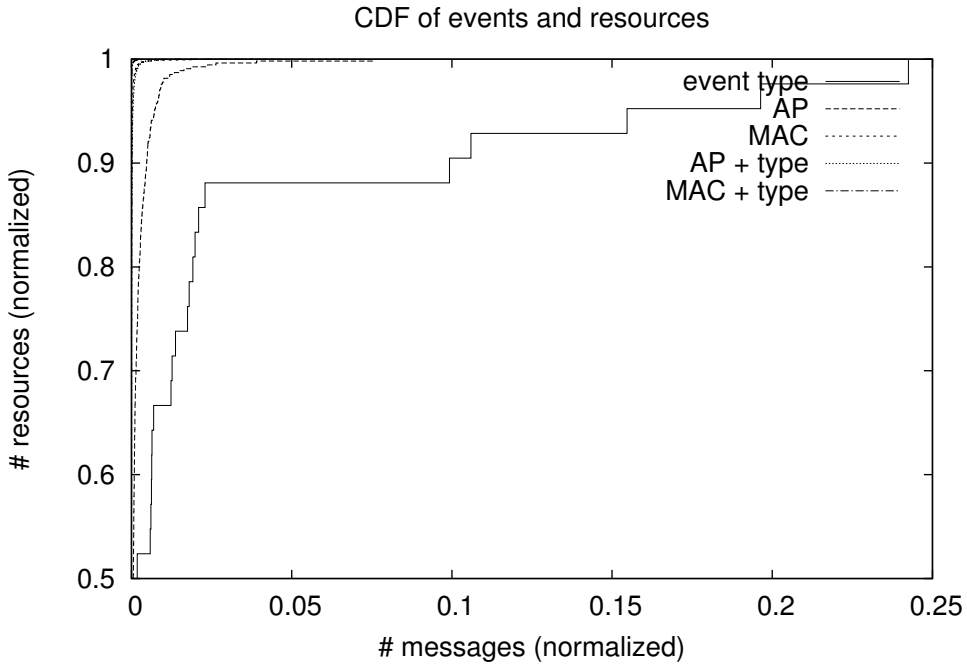

Figure 7.2: Event distribution for the Dartmouth dataset. We show the fraction of resources (y-axis) receiving at least the fraction "x" of the messages.

we consider a distribution of events based on event-type.

Distribution based on event type is used in pub-sub policy-based event systems for *filtering* events (e.g., [82]). The processing of a policy is distributed across brokers by processing the policy in groups of two event types. For example, a policy in the form $A \wedge B \wedge C \rightarrow violation$ is split into two parts: $A \wedge B \rightarrow partial_{AB}$ and $partial_{AB} \wedge C \rightarrow violation$. A broker b_1 receives events of type A and B and send to a broker b_2 the resulting events of of type $partial_{AB}$. Broker b_2 integrates the events $partial_{AB}$ with events of type C to identify all policy violations. While at first glance this splitting of the rule is similar to the one we describe in this paper, this process does not consider in the mapping the value that the variables assumes and groups events only based on their predicates (i.e., their type).

To the best of our knowledge, current pub-sub policy-based systems do not address the problem of limiting the knowledge maintained at each broker. Allocating rules and events to brokers in this solution is a challenging problem. A solution that minimizes the overall number of events would require mapping rules containing similar set of predicate types on the same

brokers. However, by doing such an allocation, the maximum knowledge on brokers becomes large, as each different rule might require a slightly different set of predicates. To provide a fair comparison, we perform an explicit allocation of type-based policies to brokers so that the knowledge in each broker is reduced. We allocate in the same broker partial rules that manage the same events, and we balance the remaining rules across brokers to balance the maximum knowledge.

The evaluation of our solution needs to analyze the behavior of the system with a wide range of policies. While there are already a limited number of policies specified in regulatory documents, the way to map these abstract policies in rules that rely on information acquired from the system depends on the organization. An evaluation that focuses only on these policies would be limited in evaluating the system for the future types of complex policies. For this reason, we evaluate our system using a set of semantically meaningful but randomly generated polices. We use the semantic relations between the events in our datasets to construct policies that preserve these relations. We create such policies by defining a graph that relates events and resources: nodes in the graph are the resources; edges of the graph are the events generated in our datasets. Each policy is constructed by a random walk through the graph. The semantic graphs we used in the evaluation of the SNMP data is shown in Fig. 7.3. We use the SNMP dataset as it provides the most diverse set of events for the construction of complex policies.

We quantify the amount of information obtained by an attacker using the number of events, as each event represents an atomic piece of information about the system. The distributed execution of the rules, both in the resource-based solution and in the type-based solution, creates partial results in form of events. These events might not carry complete information about the system's condition. For example, we can consider a resource ruleset $p_1(A, B), p_2(B, C) \rightarrow partial_B(C)$ and $partial_B(C), p_3(C, D) \rightarrow violation(C)$. The knowledge of the event $partial_B(r)$ can be used to infer that there exist two events in the form $p_1(u_a, u_b), p_2(u_b, r)$. However, the exact values for the resources u_a and u_b are unknown. Even if such information is partial, it might still be useful to an attacker. In our evaluation we consider this type of inference. As we do not know the usefulness of partial events, we consider the events that can be inferred by the knowledge of $partial_B$ as complete events. Hence, we might be overestimating the knowledge acquired by an

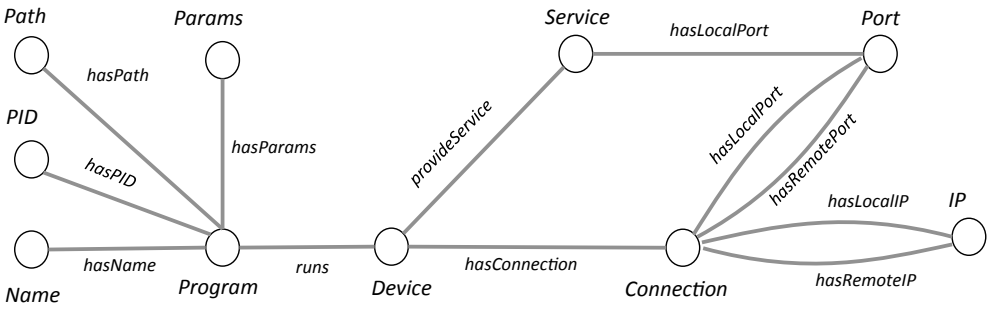

Figure 7.3: Semantic relations between resources and events. We use this graph to generate random policies that preserve a valid meaning.

attacker. This type of "backward-inference" is measured by inverting the direction of all the rules in the resource ruleset. In our example, we measure the number of events in a knowledge base containing the rules $partial_B(C) \rightarrow p_1(u_a, u_b), p_2(u_b, C)$ and $violation(C) \rightarrow p_3(C, u_d), partial_B(C)$. In this KB, the knowledge of the event $violation(C)$ allows the attacker to infer three other events. We only count these events if they are not already known by the attacker because of other information contained in the same KB.

We evaluate the ability of the system to distribute information across the brokers. For estimating the protection provided by the system against a MAX_ALL attacker we measure the number of events stored in each broker. We see that brokers, on average, store less than 10% of the events. At the most, we have one broker that stores about 17% of the events. The effects of increasing the number of rules are limited: the information about resources is reused across the execution of multiple rules. The effects of inference are also limited. By performing inference we can obtain a few additional predicates about resources in the system. We see that a type-based solution requires storing a larger number of events in brokers for all cases. Fig. 7.4 shows the normalized number of events stored in each broker when the number of rules increases.

We measure the normalized maximum number of events stored in a broker with the increase of the length of the rules. Having longer rules creates the need to match a larger number of predicate combinations. This increases the number of events to store both in the resource-based solution and in the type-based solution. However, the number of events in the resource-based solution remains significantly lower. For all cases, the maximum is obtained as the average maximum amount of events across multiple executions. These

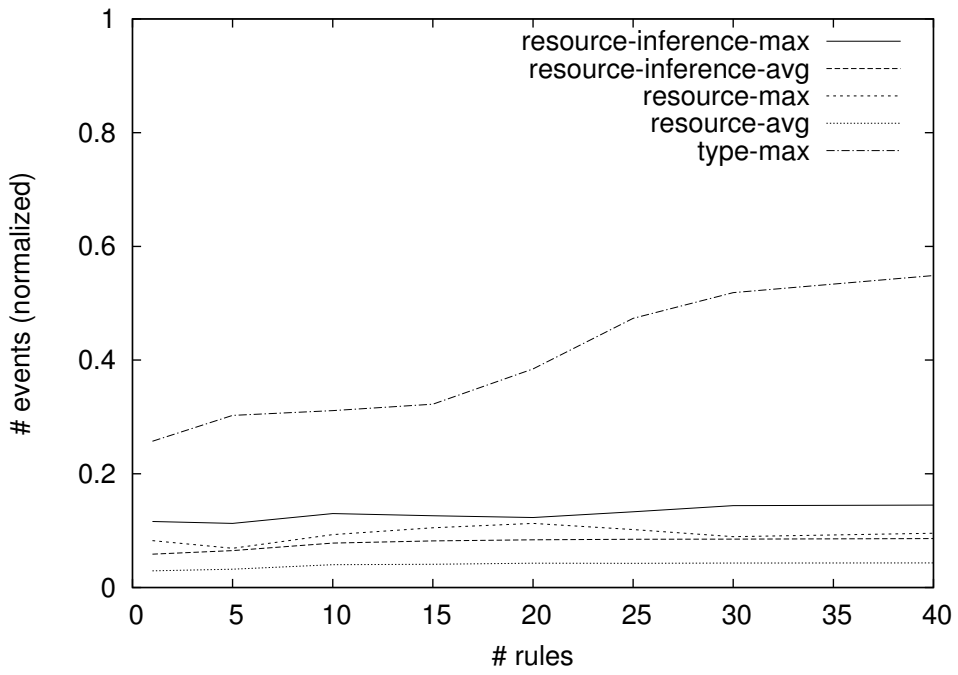

Figure 7.4: Average and maximum number of events in each broker. 20 brokers.

results are shown in Fig. 7.5.

Another measure of the protection provided toward a MAX_ALL attacker is the cumulative effects of the compromises of multiple brokers. We consider brokers in decreasing number of events to consider the worst case. We see that our resource-based approach distributes the load so that compromises of multiple brokers still have limited effects. These results are shown in Fig. 7.6.

Next, we measure the effects of our event distribution against an attacker interested in knowing all events of a specific type (i.e., a MAX_EVENT_TYPE attacker). We randomly select an event type and declare it "type-critical". We measure the maximum fraction of "type-critical events" that an attacker obtains when compromising a broker. As the event-type distribution uses event-type for its distribution, our technique provides a better protection against this type of attack. These results are shown in Fig. 7.7. We see that for a type-based distribution of events the totality of events of a specific type can be found on a specific node. Hence, the value is 1 for all cases. In our resource-based distribution, the number of type-critical events stored in each broker is limited.

We measure the effects of our approach against an attacker interested in

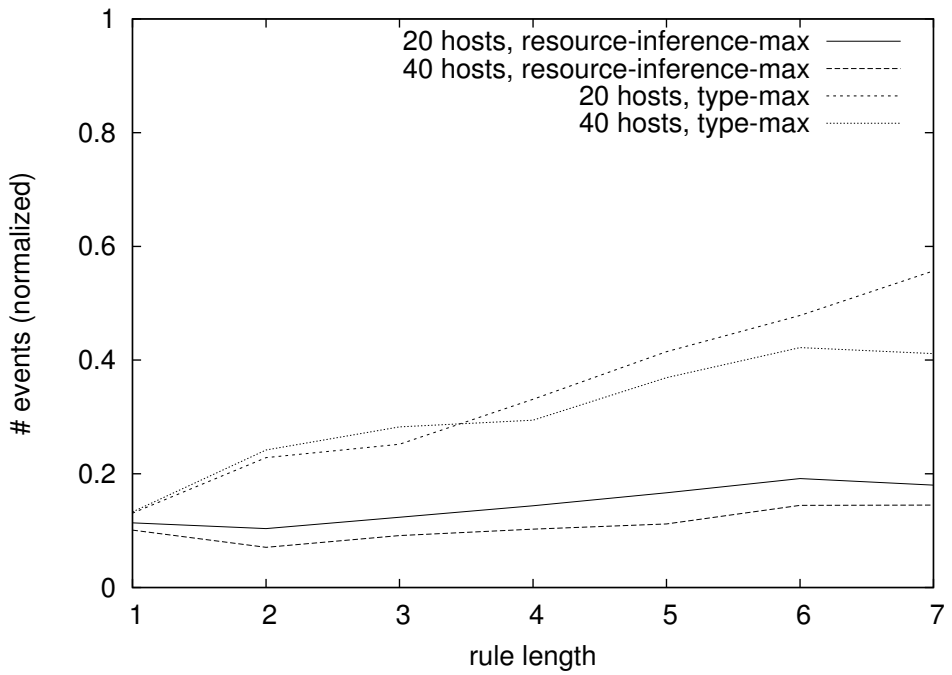

Figure 7.5: Events in each broker as a function of the length of the rule. 20 rules.

obtaining information about a specific set of resources (i.e., a MAX_EVENT_CRITICAL attacker). We randomly marked 10% of the resources as critical. We define as "resource-critical" the events that relate to one (or more) of such critical resources. Our goal is to minimize the amount of events related to critical resources acquired by the attacker. We measure the max value of critical events stored in each broker for our resource-based distribution and for an event-type distribution. We show that the resource-based distribution limits the number of critical events stored in each broker. These results are shown in Fig. 7.8.

However, for attackers targeting a specific resource, a pure resource-based mapping algorithm provides weak guarantees: an attacker only need to compromise a specific broker to access all information about the resource. A better strategy for deployment can take advantage of a mix-mechanism for distributing data. This method distributes events about a critical resource to multiple brokers instead of concentrating them in a single broker. However, it also increases the overall distribution of information, as it reduces the reuse of events across multiple rules. We show the effects of this distribution in Fig. 7.8 with the suffix *-opt*.

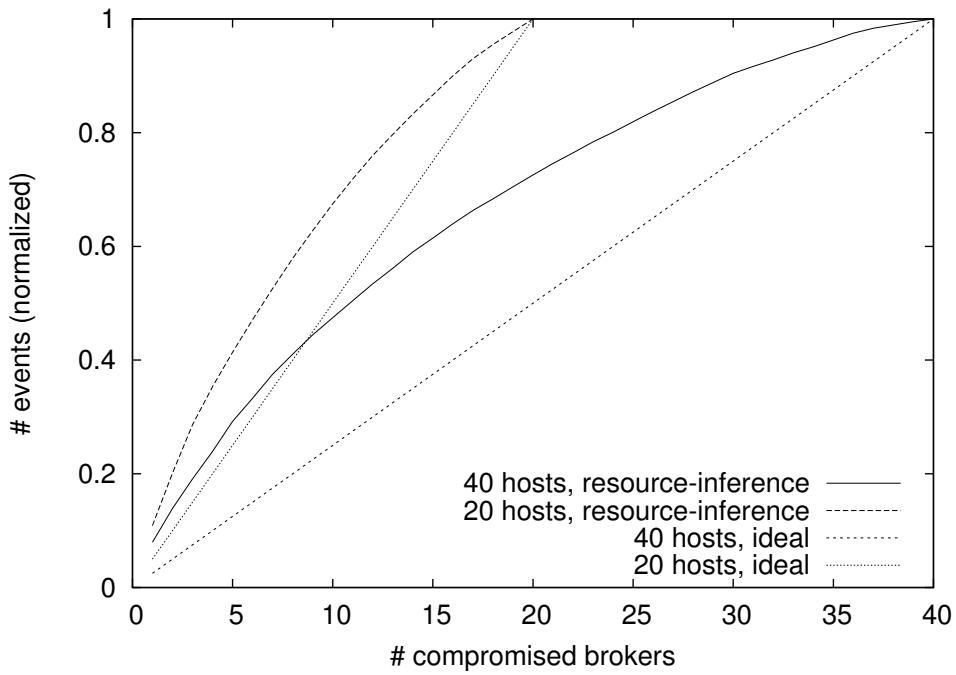

Figure 7.6: Fraction of events obtained when multiple brokers are compromised. 20 rules. The brokers with most events are compromised first.

The multi-step validation has limited negative effects on the performance of the system. The distributed correlation process adds a delay in the detection of problems. The result of the processing in one broker needs to be forwarded to other nodes before a complete detection is performed. We measure the average delay in the detection introduced by our system. We see that the delay introduced is within a second even for long rules. The slightly lower delay in the 40-host solution is created by a lower average communication delay in the 40-host network configuration of our EC2 deployment. We show this results in Fig. 7.9. The load of receiving event messages is distributed across servers in a way proportional to the amount of events stored in each broker.

In summary, our solution provides a better distribution of information and, hence, a better protection against attacks toward the confidentiality of the monitoring system than other previous solutions for the distribution of information.

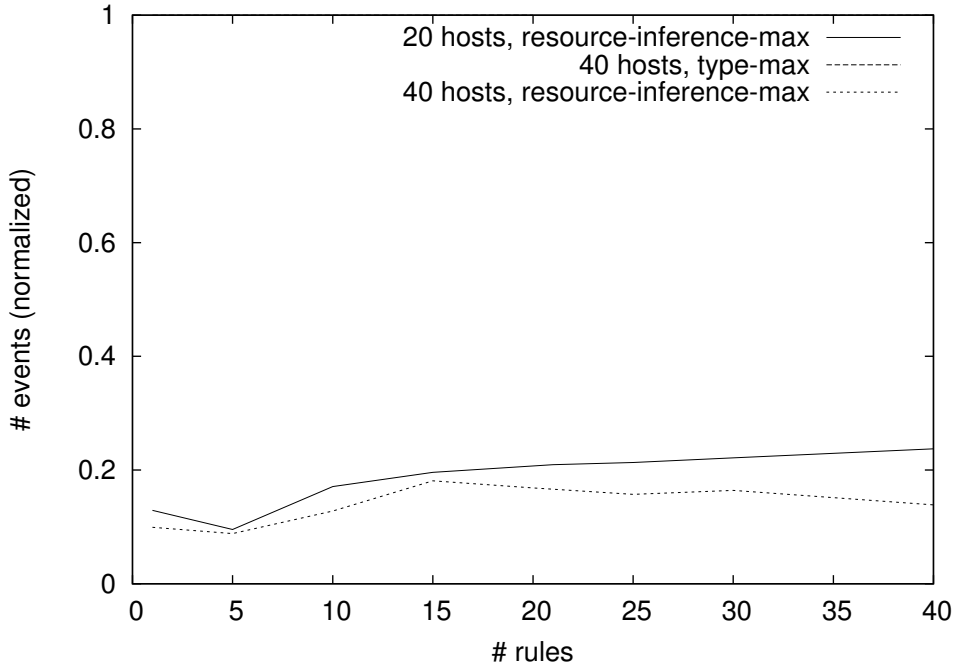

Figure 7.7: Average and maximum fraction of *type-critical* events contained in each broker.

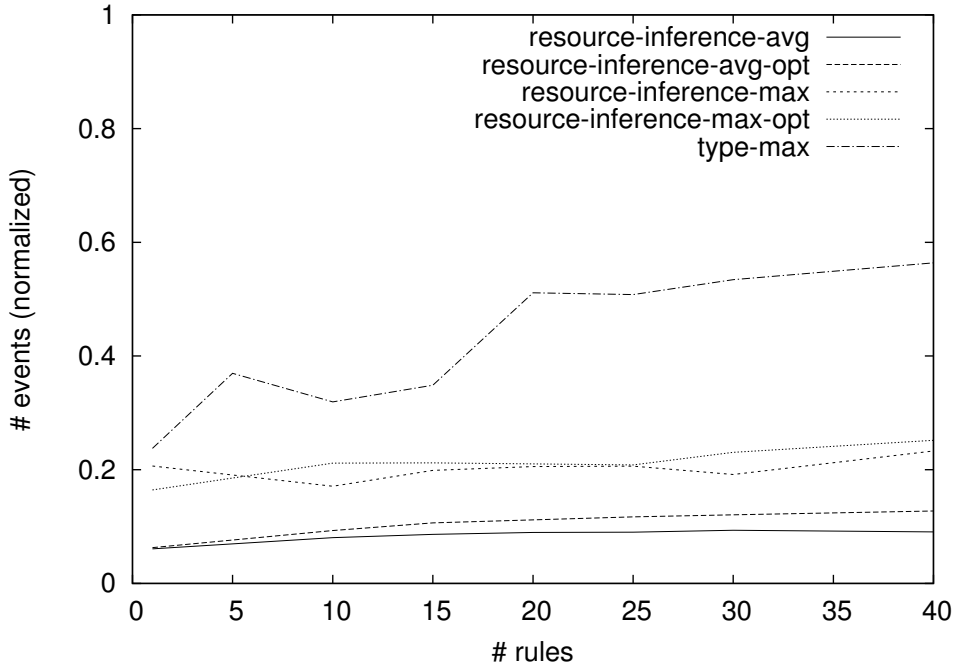

Figure 7.8: Average and maximum fraction of *resource-critical* events contained in each broker. 20 brokers.

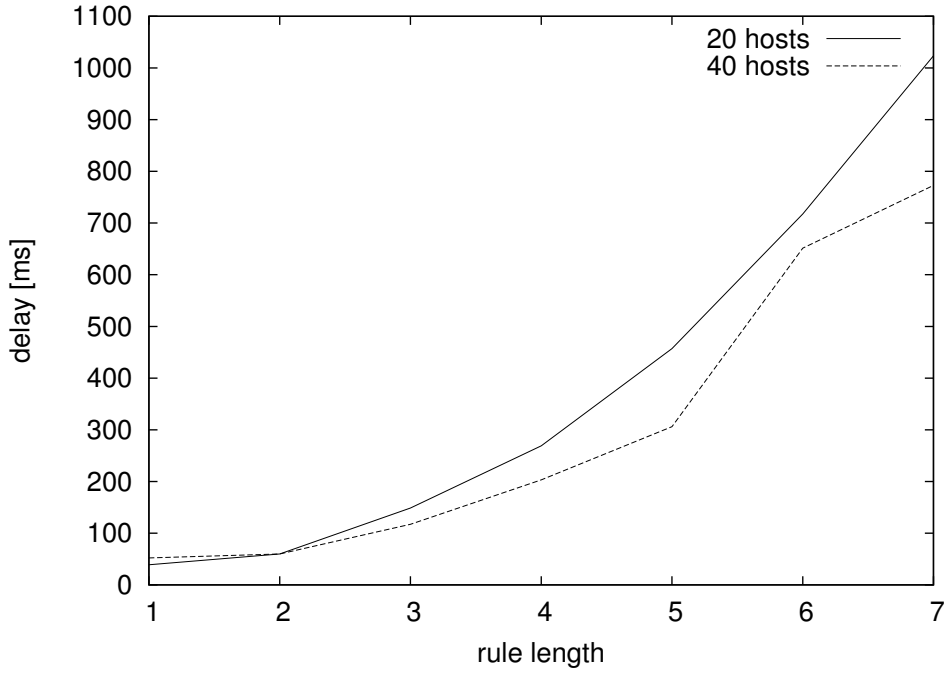

Figure 7.9: Violation detection delay introduced by our algorithm. 20 rules.

CHAPTER 8

CONCLUSION

In this thesis, we analyzed the problem of sharing data across organizations for detecting violations of security policies. We identified several scenarios from real-word policies in which such a sharing is necessary. We formalized the problem as an event-sharing problem, and we introduced two representations for events and rules: an event-based representation and a state-based representation. Based on such a framework, we computed the theoretical minimal amount of information that needs to be shared (i.e., *need-to-know*) for detecting all policy violations.

We introduced two approaches for reducing the amount of information shared to a value close to the theoretical minimum. We developed algorithms and protocols based on such approaches, and we implemented them in a novel monitoring system architecture.

The first approach uses knowledge about the *complete observability* of the information observed by each organization to determine which events should be shared. If all events necessary for validating a policy are found locally, then no sharing is necessary. Otherwise, some information is shared to ensure the detection of violations.

The second approach uses the intuition that monitoring systems collect events providing information about the state of *resources* in the system, such as computer systems, users, or software programs. By taking advantage of such an intuition, we developed a distributed reasoning algorithm that limits the interactions between organizations: the only systems sharing information are the ones managing resources potentially involved in a violation. We introduced a distributed protocol based on garbled circuits that, for simple policies, limits the amount of information shared to the theoretical minimum. We evaluated all techniques in different scenarios, and we showed that the application of our approaches reduces the amount of information shared across the organization boundaries, when compared to other solutions.

At last, we provided a security analysis of our monitoring architecture. As security monitoring is a fundamental service in modern systems, its security is fundamental to enable proper response to attacks and proper configuration of systems. We analyzed the impact of attacks on the integrity, availability, and confidentiality of the monitoring data. We showed that in many cases, our monitoring system fails gracefully without the causing catastrophic security failures of centralized systems.

Future work should focus on extending this work in several directions. Many optimizations can be introduced in our distributed reasoning algorithm. Such optimization should take into account the frequency of events and their relative important to the organization to minimize dynamically the amount of information shared.

Additionally, future work should improve the resilience of our architecture to attacks. In our current model, we assume that organizations are behaving correctly and providing a trustworthy representation of the information collected by their monitoring systems. If a monitoring system is acting maliciously, our distributed processing limits the impact of the compromise, but our architecture does not include mechanisms for validating that the information provided by an organization is, in fact, a trustworthy representation of the state of the system.

REFERENCES

[1] United State Government, "Federal Information Security Management Act (FISMA)," 2002. [Online]. Available: http://csrc.nist.gov/groups/SMA/fisma/index.html

[2] Payment Card Industry Security Standards Council, "Payment Card Industry (PCI) Data Security Standard," Tech. Rep. October, 2010. [Online]. Available: http://en.scientificcommons.org/8858188

[3] R. Pang, "A high-level programming environment for packet trace anonymization and transformation," in *ACM SIGCOMM*, Germany, 2003.

[4] A. Slagell, K. Lakkaraju, and K. Luo, "Flaim: A multi-level anonymization framework for computer and network logs," in *LISA*, 2006.

[5] P. Lincoln, P. Porras, and V. Shmatikov, "Privacy-preserving sharing and correction of security alerts," in *Proceedings of the 13th conference on USENIX Security Symposium-Volume 13*. USENIX Association, 2004. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251392 pp. 17–17.

[6] O. Goldreich, *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge university press, 2004, vol. 2.

[7] North American Electric Reliability Corporation, "NERC CIP 002-009," NERC - North American Electric Reliability Corporation, Tech. Rep., 2007.

[8] K. L. Dempsey, N. S. Chawla, L. A. Johnson, R. Johnston, A. C. Jones, A. D. Orebaugh, M. A. Scholl, and K. M. Stine, "Sp 800-137. information security continuous monitoring (iscm) for federal information systems and organizations," 2011.

[9] Splunk Inc, "Splunk." [Online]. Available: http://www.splunk.com/

[10] The Bro Project, "The Bro Network Security Monitor." [Online]. Available: http://bro-ids.org/

[11] R. Vaarandi, "Sec-a lightweight event correlation tool," in *IP Operations and Management, 2002 IEEE Workshop on*. IEEE, 2002, pp. 111–115.

[12] D. Bernstein, E. Ludvigson, K. Sankar, S. Diamond, and M. Morrow, "Blueprint for the intercloud-protocols and formats for cloud computing interoperability," in *ICIW'09*. IEEE, 2009, pp. 328–336.

[13] F. Liu, J. Tong, J. Mao, R. Bohn, J. Messina, L. Badger, and D. Leaf, "Nist cloud computing reference architecture," *NIST Special Publication*, vol. 500, p. 292, 2011.

[14] R. Boutaba and I. Aib, "Policy-based management: A historical perspective," *Journal of Network and Systems Management*, vol. 15, no. 4, pp. 447–480, 2007.

[15] D. E. Bell and L. J. LaPadula, "Secure computer systems: Mathematical foundations," DTIC Document, Tech. Rep., 1973.

[16] K. J. Biba, "Integrity considerations for secure computer systems," DTIC Document, Tech. Rep., 1977.

[17] D. F. Brewer and M. J. Nash, "The chinese wall security policy," in *Security and Privacy, 1989. Proceedings., 1989 IEEE Symposium on*. IEEE, 1989, pp. 206–214.

[18] D. Ferrailio, D. Kuhn, and R. Chandramouli, "Role based access control," in *15th National Computer Security Conference*, 1992.

[19] D. Ferraiolo, D. R. Kuhn, and R. Chandramouli, *Role-based access control*. Artech House Boston, 2007.

[20] R. J. Hayton, J. M. Bacon, and K. Moody, "Access control in an open distributed environment," in *Security and Privacy, 1998. Proceedings. 1998 IEEE Symposium on*. IEEE, 1998, pp. 3–14.

[21] A. Uszok, J. Bradshaw, M. Johnson, R. Jeffers, A. Tate, J. Dalton, and S. Aitken, "Kaos policy management for semantic web services," *Intelligent Systems, IEEE*, vol. 19, no. 4, pp. 32–41, 2004.

[22] J. Singh, L. Vargas, J. Bacon, and K. Moody, "Policy-Based Information Sharing in Publish/Subscribe Middleware," *2008 IEEE Workshop on Policies for Distributed Systems and Networks*, pp. 137–144, June 2008.

[23] D. Evans and D. Eyers, "Efficient Policy Checking across Administrative Domains," *2010 IEEE International Symposium on Policies for Distributed Systems and Networks*, pp. 146–153, 2010.

[24] D. Robinson and M. Sloman, "Domains: a new approach to distributed system management," in *Distributed Computing Systems in the 1990s, 1988. Proceedings., Workshop on the Future Trends of.* IEEE, 1988, pp. 154–163.

[25] Distributed Management Task Force, "Web Based Enterprise Management (WBEM) framework," July 2012, http://www.dmtf.org/standards/wbem.

[26] A. Shieh, E. Sirer, and F. Schneider, "NetQuery: a knowledge plane for reasoning about network properties," in *SIGCOMM*. ACM, 2011. [Online]. Available: http://portal.acm.org/citation.cfm?id=1921231 http://dl.acm.org/citation.cfm?id=1921231 p. 23.

[27] Payment Card Industry Security Standards Council, "Payment Card Industry (PCI) Data Security Standard," Payment Card Industry Security Standards Council, Tech. Rep., 2010.

[28] S. Jajodia and S. Noel, "Topological Vulnerability Analysis," in *Cyber Situational Awareness*, ser. Advances in Information Security, S. Jajodia, P. Liu, V. Swarup, and C. Wang, Eds. Springer US, 2010, vol. 46, pp. 139–154. [Online]. Available: http://dx.doi.org/10.1007/978-1-4419-0140-8_7

[29] X. Ou, W. Boyer, and M. McQueen, "A scalable approach to attack graph generation," in *Proceedings of the 13th ACM conference on Computer and communications security*. ACM, 2006. [Online]. Available: http://portal.acm.org/citation.cfm?id=1180446 p. 345.

[30] S. Narain, G. Levin, S. Malik, and V. Kaul, "Declarative Infrastructure Configuration Synthesis and Debugging," *Journal of Network and Systems Management*, vol. 16, no. 3, pp. 235–258, Oct. 2008. [Online]. Available: http://www.springerlink.com/index/10.1007/s10922-008-9108-y

[31] D. Xu and P. Ning, "Privacy-Preserving Alert Correlation : A Concept Hierarchy Based Approach," in *Computer Security Applications Conference, 21st Annual*, no. Acsac. IEEE, 2005, pp. 10–pp.

[32] B. Ribeiro, W. Chen, G. Miklau, and D. Towsley, "Analyzing privacy in enterprise packet trace anonymization," in *Proceedings of the 15th Network and Distributed Systems Security Symposium*, 2008.

[33] J. King, K. Lakkaraju, and A. Slagell, "A taxonomy and adversarial model for attacks against network log anonymization," in *Proceedings of the 2009 ACM symposium on Applied Computing*. ACM, 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1529572 pp. 1286–1293.

[34] M. Burkhart, M. Strasser, D. Many, and X. Dimitropoulos, "Sepia: Privacy-preserving aggregation of multi-domain network events and statistics," *Network*, vol. 1, p. 101101, 2010.

[35] G. Denker, A. Gehani, M. Kim, and D. Hanz, "Policy-Based Data Downgrading: Toward a Semantic Framework and Automated Tools to Balance Need-to-Protect and Need-to-Share Policies," in *Policies for Distributed Systems and Networks (POLICY), 2010 IEEE International Symposium on*. IEEE, 2010. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5630236 pp. 120–128.

[36] J. H. Huh and J. Lyle, "Trustworthy Log Reconciliation for Distributed Virtual Organisations," in *Trust '09*. Springer, 2009.

[37] J. H. Huh and A. Martin, "Towards a Trustable Virtual Organisation," in *IEEE International Symposium on Parallel and Distributed Processing with Applications*. IEEE, August 2009, pp. 425–431.

[38] David Grawrock, *The Intel Safer Computing Initiative*. Intel Press, 2006, ch. 1–2, pp. 3–31.

[39] C. M. O'Keefe, "Privacy and the use of health data - reducing disclosure risk," in *Health Informatics*, 2008.

[40] A. Lee, P. Tabriz, and N. Borisov, "A privacy-preserving interdomain audit framework," in *WPES*. ACM, 2006.

[41] Y. Deswarte, L. Blain, and J.-C. Fabre, "Intrusion tolerance in distributed computing systems," in *Research in Security and Privacy, 1991. Proceedings., 1991 IEEE Computer Society Symposium on*. IEEE, 1991, pp. 110–121.

[42] BalaBit IT Security, "Syslog-ng Logging System." [Online]. Available: http://www.balabit.com/network-security/syslog-ng/

[43] B. Schneier and J. Kelsey, "Cryptographic support for secure logs on untrusted machines," in *Proceedings of the 7th conference on USENIX*, 1998. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267549.1267553

[44] D. Ma and G. Tsudik, "A new approach to secure logging," *ACM Transactions on Storage*, vol. 5, no. 1, pp. 1–21, Mar. 2009. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1502777.1502779

[45] M. Vallentin, R. Sommer, J. Lee, and C. Leres, "The NIDS cluster: Scalable, stateful network intrusion detection on commodity hardware," in *RAID*, 2007. [Online]. Available: http://www.springerlink.com/index/x52x95424524554x.pdf pp. 107–126.

[46] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *Proc. Network and Distributed Systems Security Symposium*, 2003.

[47] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang, "D3s: Debugging deployed distributed systems," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, 2008, pp. 423–437.

[48] C. Verbowski, E. Kiciman, A. Kumar, B. Daniels, S. Lu, J. Lee, Y.-M. Wang, and R. Roussev, "Flight data recorder: monitoring persistent-state interactions to improve systems management," in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 117–130.

[49] A. Singh, P. Maniatis, T. Roscoe, and P. Druschel, "Using queries for distributed monitoring and forensics," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 4, pp. 389–402, 2006.

[50] J. E. López de Vergara, A. Guerrero, V. A. Villagrá, and J. Berrocal, "Ontology-based network management: study cases and lessons learned," *Journal of Network and Systems Management*, vol. 17, no. 3, pp. 234–254, 2009.

[51] D. L. Mills, "Internet time synchronization: the network time protocol," *Communications, IEEE Transactions on*, vol. 39, no. 10, pp. 1482–1493, 1991.

[52] S. Abiteboul, R. Hull, and V. Vianu, Eds., *Foundations of Databases: The Logical Level*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[53] J. Allen, "Maintaining knowledge about temporal intervals," *Communications of the ACM*, vol. 26, no. 11, pp. 832–843, 1983.

[54] C. L. Forgy, "Rete: A fast algorithm for the many pattern/many object pattern match problem," *Artificial intelligence*, vol. 19, no. 1, pp. 17–37, 1982.

[55] K. Walzer, T. Breddin, and M. Groch, "Relative temporal constraints in the Rete algorithm for complex event detection," *Proceedings of the second international conference on Distributed event-based systems - DEBS '08*, p. 147, 2008. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1385989.1386008

[56] S. Ceri, G. Gottlob, and L. Tanca, "What you always wanted to know about Datalog (and never dared to ask)," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 1, no. 1, pp. 146–166, 1989. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=43410

[57] E. L. Siegel and D. S. Channin, "Integrating the healthcare enterprise: A primer part 1. introduction1," *Radiographics*, vol. 21, no. 5, pp. 1339–1341, 2001.

[58] J. PROULX, "E-enabling," in *Boeing Frontiers*, Aug 2003. [Online]. Available: http://www.boeing.com/news/frontiers/archive/2003/august/i_ca1.html

[59] R. D. Apaza, "Wireless communications for airport surface: an evaluation of requirements," in *IEEE Aerospace Conference*, March 2005, pp. 1778–1788.

[60] National Institute of Standards and Technology, "National Vulnerability Database V2.2," July 2012, http://nvd.nist.gov/.

[61] F. Bry and M. Eckert, "Towards formal foundations of event queries and rules," in *VLDB*, 2007. [Online]. Available: http://www.pms.ifi.lmu.de/mitarbeiter/eckert/publications/EDA-PS2007.pdf

[62] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 382–401, July 1982. [Online]. Available: http://portal.acm.org/citation.cfm?doid=357172.357176

[63] Rice University and MPI, "FreePastry," Mar. 2009, http://www.freepastry.org/.

[64] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*. ACM, 2003. [Online]. Available: http://portal.acm.org/citation.cfm?id=945462 pp. 164–177.

[65] B. D. Payne, M. D. P. D. a. Carbone, and W. Lee, "Secure and Flexible Monitoring of Virtual Machines," *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pp. 385–397, 2007.

[66] M. Ben-Yehuda, M. Day, Z. Dubitzky, M. Factor, N. HarEl, A. Gordon, A. Liguori, O. Wasserman, and B. Yassour, "The turtles project: Design and implementation of nested virtualization," in *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI), October 2010, Vancouver, British Columbia, Canada*, 2010, pp. 423–436.

[67] M. Montanari, J. H. Huh, D. Dagit, R. Bobba, and R. H. Campbell, "Evidence of log integrity in policy-based security monitoring," in *The Second International Workshop on Dependability of Clouds, Data Centers and Virtual Machine Technology (DCDV 2012)*, 2012.

[68] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron, "SCRIBE: a large-scale and decentralized application-level multicast," *infrastructure, IEEE Journal on Selected Areas in Communications (JSAC), Vol. 20, No.*, vol. 8pp, pp. 1489–1499, 2002.

[69] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Middleware 2001*, no. November 2001. Springer, 2001. [Online]. Available: http://www.springerlink.com/index/404522p56nm85503.pdf pp. 329–350.

[70] M. Montanari, J. H. Hun, R. B. Bobba, and R. H. Campbell, "Limiting Data Exposure in Monitoring Multi-domain Policy Conformance," in *International Conference on Trust and Trustworthy Computing (TRUST)*, 2013.

[71] P. Hunt, M. Konar, F. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems," in *USENIX ATC*, vol. 10, 2010.

[72] Y. Huang, D. Evans, J. Katz, and L. Malka, "Faster secure two-party computation using garbled circuits," in *USENIX Security Symposium*, 2011.

[73] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank, "Extending oblivious transfers efficiently," *Advances in Cryptology-CRYPTO 2003*, pp. 145–161, 2003.

[74] M. Montanari, L. Cook, and R. Campbell, "Multi-organization policy-based monitoring," in *IEEE POLICY, 2012*, 2012.

[75] M. Montanari, J. Huh, D. Dagit, R. Bobba, and R. Campbell, "Evidence of log integrity in policy-based security monitoring," in *Dependable Systems and Networks Workshops (DSN-W), 2012*. IEEE, 2012.

[76] Y. Huang, J. Katz, and D. Evans, "Quid-pro-quo-tocols: Strengthening semi-honest protocols with dual execution," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 272–284.

[77] M. Montanari and R. Campbell, "Confidentiality of event data in policy-based monitoring," in *Dependable Systems and Networks (DSN), 2012*. IEEE, 2012.

[78] Diginotar, "DigiNotar reports security incident." [Online]. Available: http://www.vasco.com/company/ press_room/news_archive/ 2011/news_diginotar_ reports_security_incident.aspx

[79] C. Tankard and D. Pathways, "Persistent threats and how to monitor and deter them," *Network Security*, vol. 2011, no. 8, pp. 16–19, 2011. [Online]. Available: http://dx.doi.org/10.1016/S1353-4858(11)70086-1

[80] B. Sangster, T. O'Connor, T. Cook, R. Fanelli, E. Dean, W. Adams, C. Morrell, and G. Conti, "Toward instrumenting network warfare competitions to generate labeled datasets," in *Proceedings of the 2nd conference on Cyber security experimentation and test*. USENIX Association, 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855490 pp. 9–9.

[81] D. Kotz, T. Henderson, I. Abyzov, and J. Yeo, "CRAWDAD data set dartmouth/campus (v. 2009-09-09)," 2009.

[82] E. Fidler, H. Jacobsen, and G. Li, "The PADRES distributed publish/subscribe system," *Feature Interactions in Telecommunications and Software Systems*, 2005.