AUTOMATIC VERIFICATION OF SECURITY POLICY IMPLEMENTATIONS

BY

SANKALP SINGH

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, 2012

Urbana, Illinois

Doctoral Committee:

Professor William H. Sanders, Chair and Director of Research
Professor Roy H. Campbell
Associate Professor Darko Marinov
Professor David M. Nicol

ABSTRACT

Networked systems are ubiquitous in our modern society. They are found in settings that vary from mundane enterprise IT systems to critical infrastructure systems. The security of networked systems is important given their widespread use. In particular, the emerging scenarios and the likely trends for the future of critical networked systems make the security of those systems a paramount concern, especially in the area of controlling access to the critical elements of the system over communication networks; successful cyber-attacks on such systems, in a worst-case scenario, could result in loss of life, or in massive financial losses through loss of data, actual physical destruction, misuse, or theft.

Access control is a cornerstone of network security. In a modern networked system, access control is implemented through a variety of devices and mechanisms that include, but are not limited to, router-based dedicated firewalls; host-based firewalls, which could be based in software or hardware; operating-system-based mechanisms, such as the mandatory access control in the National Security Agency's (NSA's) SELinux; and middleware-based mechanisms, such as the Java Security Manager. Such devices and mechanisms collectively implement a networked system's global policy (which is usually implicit), which specifies the overall system-level objectives with respect to resource access. However, it has been shown in empirical studies that misconfiguration of access control enforcement points is common. The problem of identifying those misconfigurations is compounded when several such mechanisms are present, as the complex interactions among those distributed and layered mechanisms can mask problems and lead to subtle errors.

In this dissertation, we propose a framework for performing comprehensive security analysis of an automatically obtained snapshot of an access control policy implementation (e.g., firewall rule-sets) to check for compliance against a (potentially partial) specification of the global access policy. We identify and classify possible errors that can be found in global policy implementations,

including both policy violations and internal inconsistencies. We provide detailed formalisms that can be used to efficiently model the topology of the networked system being analyzed and the rule-sets from multiple types and makes of firewalls that may be present on the network. The formalisms are XML-based, with sound mathematical underpinnings. We present an XML-based global policy specification language, with algorithms that ensure internal consistency of specifications written in that language and resolve any conflicts. We show that our specification language is at least as expressive as linear temporal logic.

We describe an efficient algorithm for exhaustive analysis to identify all the inconsistencies and policy violations. The analysis algorithm utilizes specialized data structures, that we call *multi-layered rule-graphs*, to dramatically improve performance. We provide additional mechanisms for identifying the root causes of any problems discovered. We further enhance the scalability of our analysis by presenting an algorithm for statistical analysis of the networked system; the algorithm uses importance sampling, and produces a sample set of violations and a quantitative estimate of the remainder.

To facilitate the analysis, our framework includes techniques that automatically infer the network topology for the system being analyzed based simply on the firewall rule-sets implemented. The framework has been implemented with a sophisticated graphical front-end as the Network Access Policy Tool (NetAPT). We demonstrate the efficiency, scalability, and extensibility of our techniques through analytic evaluation and empirical evidence based on real-world testing. We also present an algorithm for automatically generating a benchmark suite for testing NetAPT; the algorithm learns the defining characteristics of our real-world data sets and generates random networks and firewall rule-sets that are representative of the real-world ones.

To my parents, for their love and support.

ACKNOWLEDGMENTS

The road I have taken in my journey through graduate school has been long and winding. There were times when the end did not seem to be in sight, but ultimately, I am glad I persevered and find myself tremendously richer for the experience. I could not have completed this journey without the support and encouragement of many people, and I would like to thank them.

First, I would like to thank my advisor, Professor William H. Sanders, for his technical, financial, and, when needed, emotional guidance and support throughout my graduate career. I am grateful for the opportunity to learn the fundamentals of good and high-impact research from him. I will always be thankful for his unwavering support when extraneous factors introduced delays in my thesis research. I would like to thank Professor David M. Nicol for serving as my unofficial co-advisor, and for serving on my thesis committee. His technical insights and guidance have helped me mold and sharpen into focus a lot of the research presented in this dissertation. I would like to thank both Professor Sanders and Professor Nicol for allowing me the opportunity and freedom to tackle relevant and meaningful problems.

I would like to thank Mouna Bamba for helping me to bring my research out from the ethereal into the corporeal. It has been an incredibly pleasant and instructional experience collaborating with her on the NetAPT tool, and I am grateful for all the support and encouragement she has provided.

I would like to thank Edmond Rogers for sharing his technical insights and wisdom that can only come through experience. His help and feedback have been invaluable in refining the research presented in this dissertation.

I am thankful to Jenny Applequist for the education in grammar she has given me, and the many long hours of work she has graciously spent helping me with my writing.

I would like to thank the members of the PERFORM group, who have been like a family to me

during graduate school. I am thankful for the nourishing and intellectually simulating environment the PERFORM group has provided, and I am not surprised that I have formed some of my closest friendships within the group. I am grateful to Eric Rozier and Ryan Lefever for their friendship; their support and mentorship have been invaluable. I would like to thank the following members for their support and friendship: Adnan Agbaria, Sobir Bazarbayev, Robin Berthier, Shuyi Chen, Tod Courtney, Michel Cukier, David Daly, Nathan Dautenhahn, Salem Derisavi, Doug Eskins, Ahmed Fawaz, Michael Ford, Shravan Gaonkar, Mark Griffith, Vishu Gupta, Michael Ihde, Kaustubh Joshi, Ken Keefe, Vinh Lam, Elizabeth LeMay, James Lyons, Michael McQuinn, Hari Ramasamy, Fabrice Stevens, and Saman Aliari Zonouz.

I thank Professors Roy H. Campbell and Darko Marinov for serving on my committee and offering me their support, guidance, and suggestions. Their feedback during my preliminary examination was critical to my thesis, and their support throughout this process helped me to shape my research into a successful dissertation.

I would also like to thank the various agencies[1] that funded my research. This material is based in part upon work supported by the U.S. Department of Homeland Security under Grant Award Number 2006-CS-001-000001; by the Institute for Information Infrastructure Protection (I3P) under grant 2003-TK-TX-0003 from the Office of Domestic Preparedness and the Office of Science and Technology of the U.S. Department of Homeland Security; by the Department of Energy under Award Number DE-OE0000097; by the National Science Foundation under Grant No. CNS-0524695; by DARPA under contract number F30602-02-C-0134; by Motorola, Inc.; and by the Boeing Company. Any opinions, findings, points of view, and conclusions or recommendations expressed in this document are those of the author and do not necessarily reflect the official position or views, either expressed or implied, of the National Science Foundation, the U.S. Department of Homeland Security, the Office of Domestic Preparedness, the Office of Science and Technology, or the other supporting agencies.

I would like to thank my family for all of their love, support, and encouragement. I really

[1] This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

appreciate their encouragement and patient support as I made my circuitous way through graduate school. I thank my parents for their support of my academic pursuits throughout my life and nurturing of my interests. I thank my brother Tanmay for sharing so much of my life and being a great friend in addition to being a great brother.

Last but certainly not least, I want to thank God, without whom none of this would be possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AAA	Authentication, Authorization, and Accounting
ACL	Access Control List
ASA	Adaptive Security Appliance
CTL	Computation Tree Logic
DMZ	DeMilitarized Zone
DTD	Document Type Definition
LTL	Linear Temporal Logic
NetAPT	Network Access Policy Tool
NAT	Network Address Translation
NSA	National Security Agency
PCS	Process Control System
PIX	Private Internet eXchange
SCADA	Supervisory Control And Data Acquisition
SAML	Security Assertion Markup Language
SELinux	Security-Enhanced Linux
XACML	eXtensible Access Control Markup Language
XML	eXtensible Markup Language

CHAPTER 1

INTRODUCTION

Networked systems are used in a large number of settings, including many critical infrastructure systems, such as chemical plants, electric power generation and distribution facilities, water distribution networks, and waste water treatment facilities. The emerging scenarios and the likely trends for the future of critical networked systems demand that the problem of securing these systems receive immediate attention, especially in the area of controlling access to the critical elements of the system over communication networks. Given the mission-critical nature of a significant number of large networked information systems, it is extremely important to ensure their protection against cyber-attacks, which, in a worst-case scenario, could result in loss of life, or in massive financial losses through loss of data, actual physical destruction, misuse, or theft.

A modern networked system includes a variety of devices and mechanisms for controlling access to its resources. These access control mechanisms include, but are not limited to, router-based dedicated firewalls; host-based firewalls, which could be based in software or hardware; operating-system-based mechanisms, such as the mandatory access control in the National Security Agency's (NSA's) SELinux; and middleware-based mechanisms, such as the Java Security Manager, that provide for specification and enforcement of fine-granularity access control policies for Java programs.

The importance of correctly implementing access control for effective intrusion prevention cannot be overestimated. A survey of the SANS Institute's top 20 vulnerabilities [1] shows that a significant number of them are defended against by appropriate configurations of access control policy. To defend systems against the most critical known threats, one has to be able to validate security policy implementation. However, distributed and layered mechanisms, such as those listed above, can interact in complex ways that can lead to subtle errors and mask problems. It can be difficult to discern the global picture that emerges from the local configurations of these myriad ac-

cess control elements. As a result, it is not surprising that misconfigurations of these mechanisms are a major source of security vulnerabilities. In fact, a study in 2004 suggested that most firewalls (the most popular access control mechanism) suffer from misconfigurations [2]. It is important for the administrators of computer networks to have ways to make sure that high-level specifications of such system access constraints are reflected in the actual configurations of the access control mechanisms spread throughout the system. Furthermore, if the implementation of policy (device configurations) is not in compliance with the specification, a diagnosis to locate the root causes of the problem is critical.

This dissertation describes a framework and a set of constituent techniques to address the needs described above. Our techniques allow for the analysis of the security policy *implementation* for conformance with the global security policy *specification*. They integrate policy rules (i.e., configuration information) from a large variety of sources typically found in a modern network and provide for a detailed offline analysis, as well as dynamic online analysis of incremental configuration changes that allows for detection of policy implementation holes during operation. They ensure scalability with increasing network size and complexity via a statistical analysis mode. We have implemented the framework in the form of a tool, the Network Access Policy Tool (NetAPT), which includes a graphical front-end for usability and ease of information management.

1.1 Thesis Statement

It is difficult to meet best-practices recommendations and compliance requirement rigorously by hand without significant person-hour investment. There exists a need for methods that can express best practices and compliance requirements for large networked systems as global access policy in a machine-checkable form. Furthermore, we need techniques that can automatically detect and identify violations of the global access policy by its implementation (in the form of configuration of firewalls and other access policy enforcement devices).

It is our thesis that:

- Access control policy *implementation* in networked systems can be verified for compliance against a (potentially partial) *specification* of the global access policy.

- Mechanisms for such verification can be demonstrably efficient, scalable and extensible, and require minimal user guidance.
- Those mechanisms can be of use to real system administrators and auditors by helping them identify security problems that would otherwise be difficult to detect without a considerable time investment.

1.2 Contributions

The main contributions of this dissertation are as follows:

- A classification of the possible errors that can be found in global policy implementations. This includes errors leading to policy violations and/or internal rule-set inconsistencies.
- An XML-based global policy specification language with mechanisms for direct transcription as well as translation from other formalisms, such as Linear Temporal Logic (LTL) and GUI-based specifications. We provide algorithms for ensuring internal consistency of the specification and resolution of any conflicts. We prove that our specification language is at least as expressive as LTL.
- A unified XML-based schema for representing rule-sets from multiple types and makes of firewalls. We currently support nearly full feature-sets of the devices from the four most popular commercial firewall manufacturers.
- A multi-layered rule-graph data structure for efficient modeling of network topology and the accompanying access policy implementation. We use multidimensional interval trees (continuous elements) and hash-tables (discrete elements) to represent rule-graphs, policy constraints, and traffic attribute sets.
- An efficient algorithm for exhaustive analysis that produces a complete list of inconsistencies and policy violations. We provide additional mechanisms for identifying the root causes of the reported violations.

- An algorithm for statistical analysis of the networked system, which produces a sample set of violations and a quantitative estimate of the remainder.

- A method for automatically inferring the network topology from the configuration of the layer 3 devices (firewalls, routers, and switches).

- An implementation of the proposed framework in the form of the Network Access Policy Tool (NetAPT). The tool provides a high-functionality GUI as well as a powerful command-line interface. It incorporates implemented policy rules from a wide variety of sources, and, through extensive automation, minimizes the need for user guidance. It has been used successfully at several real-world sites.

- A method for learning the defining characteristics of a set of network topologies and the rule-sets of the firewalls therein, and use them to generate random networks and firewall rule-sets that are similar. The generated networks can be varied along dimensions such as numbers of sub-networks, access control devices, hosts, applications deployed, and rule-set size and complexity. We use the randomly generated networks to provide comprehensive experimental results demonstrating the efficacy and scalability of our analysis and topology-inference algorithms. The experimental results supplement the analytic evaluation of those algorithms.

1.3 Organization

The remainder of this dissertation is organized as follows.

Chapter 2 begins with some background information on how access control is implemented in modern networked systems. We then provide a detailed and comprehensive classification of possible errors that can be found when implementing access control through firewalls. We include both errors due to internal inconsistencies in the firewalls' rule-sets and errors resulting from violations of the user-specified global policy.

Chapter 3 presents the state of the art of the research in related areas, particularly policy specification and identification of misconfigurations in firewalls. We look at academic research and

several commercial products. We critically examine the prior art and identify the key challenges that still need to be addressed in order to satisfy our thesis statement.

Chapter 4 introduces the formalisms and data structures that we use to develop efficient analysis algorithms. We provide formalisms for global access policy specification, access control elements, and network topology. We describe the multilayered rule-graph data structure that we use to model the network topology and access control elements (firewall rule-sets). We detail mechanisms for efficient traversal of the rule-graph data structure using multidimensional interval trees. We conclude the chapter with a comparison of our policy specification language and LTL, proving that our specification language is at least as expressive as LTL.

Chapter 5 details our algorithms for efficient, exhaustive analysis of a given network system (topology and rule-sets) for compliance with a user-specified global access policy. We formally describe an algorithm for identifying all the internal inconsistencies in the rule-sets, an algorithm for identifying all the paths through the network that violate some tenets of the global access policy, and, finally, algorithms for identifying the root causes that result in all the violating paths. We conclude with an analytic evaluation (complexity analysis) of the algorithms described in the chapter.

Chapter 6 details importance sampling-based algorithms for statistical analysis for checking compliance. We provide descriptions of the required metrics and mathematical formulation to support the use of importance sampling. We also outline an algorithm for determining the likelihood of there being no violations if none are found when some stopping criterion is reached.

Chapter 7 describes the techniques we use for inferring the network topology from the configuration files of the firewalls and layer 3 devices (such as routers and switches). We highlight the ways in which we have made major extensions to an existing inference framework to implement our techniques. We describe the inference algorithms in detail in Appendix A.

Chapter 8 details the implementation of our framework in the form of the Network Access Policy Tool (NetAPT). We describe the tool architecture and highlight some of the tool's essential features.

Chapter 9 details experimental evaluation of our algorithms. We evaluate NetAPT on test cases drawn from various industry collaborations, and demonstrate that our algorithms are efficient and provide results that can easily be used by system administrators to identify and correct any prob-

lems with the configurations of their firewalls. We also detail a mechanism for generating a benchmark suite of automatically generated network topologies and firewall rule-sets that share essential characteristics with the data set obtained through industry collaboration.

Finally, Chapter 10 concludes with a summary of our contributions and brief discussion of future avenues for research.

CHAPTER 2

ACCESS CONTROL IN NETWORKS

Access control has long been the linchpin of system security; modern systems have multiple access control methodologies, different security models, and separate configurations for each methodology and each device. Together these all form the access control *implementation*. However, only very limited technology exists to answer crucial questions about precisely what security posture is produced, how the different access control policies interact, and whether the implementation is in compliance with an overall statement of global access control policy, among others.

To better appreciate the issue, let's examine some of the access control components of a distributed system. Firewalls are critical assets in the protection of a network. A firewall is configured through its rules (collectively referred to as a *rule-set*), which it can use to shape the traffic that crosses it. A firewall matches the incident network traffic against its rules, using traffic characteristics such as the source of origin, intended destination, and communication protocol, and either forbids or allows the traffic to pass through depending on the action indicated by the matching rule. A typical setting usually contains a distributed firewall implementation, wherein traffic may need to pass through more than one firewall to transit the network. Firewalls can be used to divide the network into secure zones that limit user and application access between zones. For example, Figure 2.1 shows a real-life network we have studied, developed as part of a large Defense Advanced Research Projects Agency (DARPA) project, which is composed of multiple network zones (in boxes) isolated by devices that enforce access control policy. Of particular interest is the fact that there are eight separate zones and over 50 different policy enforcement devices (including SELinux on some of the hosts).

Most companies configure firewalls with a "deny all, permit none" setting with a few exceptions. However, in reality, there are no generic rules that fit all situations, and the specific architecture of the system (e.g., the use of demilitarized zones (DMZ)) dictates how the rules should be configured

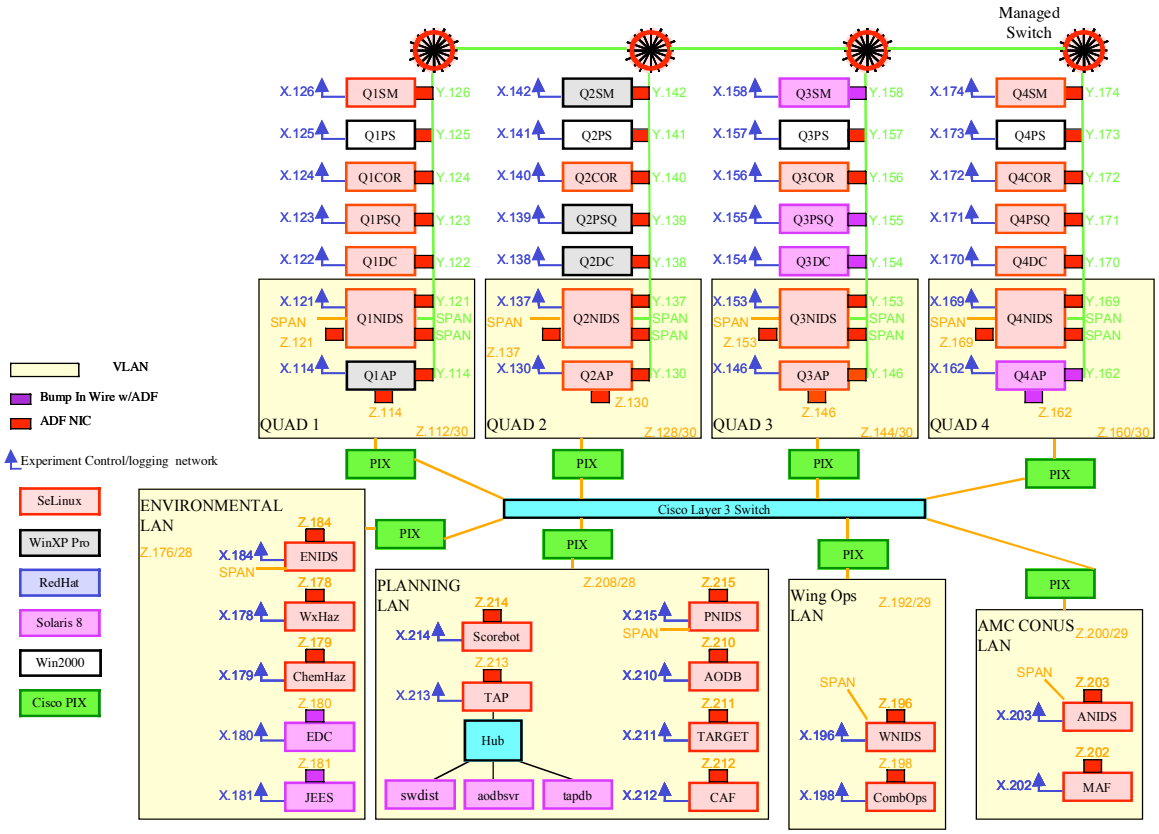

Figure 2.1: Representative Network

for effective security at a particular site. Firewalls may further support "stateful" rules that keep track of roles in a TCP/IP session and only allow traffic that is consistent with the past history of the session indicated in the network packet. In addition, firewalls may provide network address translation (NAT), which creates a mapping between the IP addresses and port numbers used in the interior of the network and a (common) IP address and port number offered to the outside network.

A system can also have host-based firewalls, implemented in either software or hardware. Software-based firewalls, such as iptables [3] in Linux and several commercial ones available for Windows, are implemented in the host operating system. Hardware-based firewalls, such as 3Com's Embedded Firewall (EFW) PCI card [4], are implemented on the network interface card (NIC) itself, and as a result, those firewalls are tamper-resistant to cyber-attackers who were able to gain control of the operating system on a host.

There are published guidelines (e.g., the National Infrastructure Security Coordination Center (NISCC) guide to good practices in firewall deployment [5]) to facilitate the development of unique

rule-sets for the different firewalls at an operating site, but they are fairly generic and have to be customized by network system administrators to the specific needs of their sites.

However, access control is more complex than just firewall rules. A finer-grained access control is provided by mechanisms available on the operating system and middleware levels. They might include discretionary in-built mechanisms for managing permissions for local objects (such as files) and privileges of the processes that are present in the default installations of popular operating systems like Linux and Windows. However, in security-conscious settings, especially for networks managing critical infrastructure, hardened versions of the traditional operating systems have been increasingly adopted. These include SELinux [6] (developed by the NSA), which is a secure version of the Linux operating system. It can be configured using a (large) number of policy files that can be compiled into the Linux kernel. It makes it possible to restrict the processes on a machine so that they run within *domains* that are isolated from each other, with explicit rules in the policy files governing transition between domains (i.e., change in privilege). It can control the ability of applications and user classes/roles to access not just file objects, but also network sockets and other applications. Policies are typically specified using type enforcement and role-based access control mechanisms. Similar functionality for Windows operating systems can be provided through third-party software, such as the Cisco Security Agent [7].

An increasing number of handheld and embedded devices that liaison with networked systems are based on, or at least support, Java. The Java Security Manager provides fine-grained access control similar to SELinux for the Java Virtual Machine, and adds another avenue for access policy implementation.

However, access control is more complex than just firewall rules. In security-conscious settings, hardened versions of traditional operating systems have been adopted. These include SELinux [6] (developed by the NSA), which is a secure version of the Linux operating system. Similar functionality for Windows operating systems can be provided through third-party software, such as the Cisco Security Agent [7]. Such software can provide mandatory mechanisms such as role-based access control, type checking, and multi-level security models. Such tightly controlled access control is likewise at the heart of the emerging "trusted network" architectures [8], for which a device must prove to the system that it meets a specification for joining the network. Once admitted, a device's access control profile is often driven by role-based access control [9].

One can use a global policy to specify at a system level the overall objectives with respect to access control. It specifies both what connectivity between roles and devices is inadmissible, and what connectivity must be supported. The rules are stated in terms of sets of roles and devices, rather than individual ones. For example, a rule might read, "An account manager in the sales network zone must be able to use sftp to forward any file in the Monthly Sales directory to his or her Reports directory, on the sales server found in the management network zone." Another rule could be "Aside from the Library Historian, which may receive SQL updates from the mirrored Historian within the process control zone, no actor on any host in the DMZ may connect to any host in the process control zone." Clearly, multiple access control policies are at play here; it is also clear that the rules can be stated in a form that allows a program that is analyzing access control configurations to determine whether the rule is satisfied.

The critical problem this work addresses is that networked systems check only fine-grained local access policy rules at single devices, not global policy. Global access policy implementations may or may not comply with global policy requirements. Without a way to check that compliance, serious security vulnerabilities can and do exist in real implementations of critical systems, causing them to fail in potentially harmful ways when attacked.

More formally, we now discuss a classification of the problems resulting from misconfigurations of access control mechanisms, in particular firewalls.

2.1 Policy Violations

Administrators often have high-level policies describing what access to the network should be prohibited (*blacklists*) or ensured (*whitelist*). It is crucial that firewall configurations exactly reflect the security policy. Any nonconforming configurations may result in undesired blocking, unauthorized access, or even the potential for an unauthorized person to alter security configurations. Therefore, a firewall must be verified against the policy.

Although policy definition is specific to individual institutions, the network security community has some well-understood guidelines on firewall configurations. From an external auditor's point of view, Wool [2] studied 37 configurations of the Check Point's FireWall-1 product and noticed

12 common firewall configuration errors. For example, 90% if the configurations allowed "any" destination on outbound rules, and "any" service on inbound rules. Allowing of NetBIOS and Portmapper/Remote Procedure Call service was another common class of errors that expose the network to very insecure services. A large number of firewalls were not configured correctly to provide proper protection. Approximately 46% of the firewalls are not configured with stealth rules to hide themselves, and over 70% of them were open to insecure management protocols or external management machines. All such "errors" affect the security of the entire network and must be carefully checked.

Another source of input for the blacklist is the bogon list [10], which describes IP blocks or port numbers not currently allocated by the Internet Assigned Numbers Authority (IANA) and Regional Internet Registries (RIRs) plus those reserved for private or special use. Attackers often use these IP blocks or ports for DoS attacks, spamming, or hacking activities. Most firewall administrators would want to ensure that traffic to and from these IP blocks and port numbers is neither explicitly nor implicitly allowed to reach their networks.

2.2 Rule-Set Inconsistencies

Firewall configurations represent the administrators intention, which should be consistent. Therefore, inconsistencies are often good indicators of misconfigurations. Checking for inconsistencies is solely based on the configuration files and does not need external input. Inconsistencies happen at three levels: intra-firewall, inter-firewall, and cross-path.

2.2.1 Intra-Firewall Inconsistencies

1. *Shadowing* refers to the case where *all* the packets that one rule intends to deny (or accept) have been accepted (or denied) by preceding rules. This often reveals a misconfiguration and is considered an "error." A rule can be shadowed by one preceding rule that matches a superset of the packets. In Table 2.1, rule 4 is shadowed by rule 2, because every UDP packet from 172.16.1.0/24 to 192.168.1.0/24 is accepted by rule 2, which matches any UDP packets destined for 192.168.1.0/24. Alternatively, a rule may also be shadowed by a set of

1	deny tcp 10.1.1.0/25 any
2	accept udp any 192.168.1.0/24
3	deny tcp 10.1.1.128/25 any
4	deny udp 172.16.1.0/24 192.168.1.0/24
5	accept tcp 10.1.1.0/24 any
6	deny udp 10.1.1.0/24 192.168.0.0/16
7	accept udp 172.16.1.0/24 any

Table 2.1: Inconsistent Single Access Control List (ACL)

rules *collectively*. For example, rule 5 is shadowed by the *combination* of rules 1 and 3. Rule 1 denies TCP packets from 10.1.1.0/25, and rule 3 denies TCP packets from 10.1.1.128/25. Collectively, they deny all TCP packets from 10.1.1.0/24, which are what rule 5 intends to accept.

2. *Generalization* refers to the case where a subset of the packets matched to this rule has been excluded by preceding rules. It is the reverse of shadowing and happens when a preceding rule matches a subset of this rule but takes a different action. In Table 2.1, rule 7 is a generalization of rule 4 because UDP packets from 172.16.1.0/24 and to 192.168.1.0/24 form a subset of UDP packets from 172.16.1.0/24 (rule 7), yet the decision for the former is different from the latter.

3. *Correlation* refers to the case where the current rule intersects with preceding rules but specifies a different action. The predicates of these correlated rules intersect, but are not related by the superset or subset relations. The decision for packets in the intersection will rely on the order of the rules. Rules 2 and 6 are correlated with each other. The intersection of them is "udp 10.1.1.0/24 192.168.1.0/24," and the preceding rule determines the fate of these packets.

Generalization and correlation are not necessarily errors, as they are commonly used techniques for excluding part of a larger set from a certain action. Proper use of these techniques could result in fewer rules. However, the techniques should be used very consciously. Access control lists (ACLs) with generalizations or correlations can be ambiguous and difficult to maintain. If a preceding rule is deleted, the action for some packets in the intersection will change. For a large and evolving list of rules, it may be difficult to remain aware of all the related generalizations and correlations when

working manually. Without a priori knowledge about the administrator's intention, we cannot be certain of whether they represent misconfigurations. Therefore, we classify them as "warnings."

2.2.2 Inter-Firewall Inconsistencies

Inconsistencies among different firewalls might not be errors. When a few firewalls are chained together, a packet has to survive the local filtering actions of all the firewalls on its path to its destination. Therefore, a downstream firewall can often rely on an upstream firewall to achieve policy conformance and can be configured more loosely. On the other hand, a downstream firewall at the inner perimeter often needs a tighter security policy. Without input from the administrator, the only inter-firewall inconsistency that can be classified as an "error" is shadowed accept rules. By explicitly allowing certain predicates, we infer that the administrator intends to receive the relevant packets.

2.2.3 Cross-Path Inconsistencies

As discussed earlier, multiple data paths could exit from the Internet to the same protected network. *Cross-path inconsistency* refers to the case where some packets denied on one path are accepted through another path. Depending on the underlying routing table, these anomalies may be exploitable. However, attacks that affect routing protocols do exist and an attacker only needs to succeed once. Cross-path inconsistencies may also be indicated by intermittently disrupted services. Routing changes may necessitate that the packets originally reaching a network switch over to another path, but the firewall rules on the new path may deny such packets. Furthermore, the second path may not always be available since the actual path is determined by the underlying routing protocol. However, routing is designed to be adaptive to link failures and heavy load. In addition, it is relatively easy to inject false routing messages [11]. A safe firewall configuration should not rely on dynamic routing constraints, and should assume that all paths are topologically possible.

Checking cross-path inconsistencies through active testing is very difficult. It may disrupt the production network since routing tables must be altered to test different scenarios. Manual auditing

1	accept tcp 192.168.1.1/32 172.16.1.1/32
2	accept tcp 10.0.0.0/8 any
3	accept tcp 10.2.1.0/24 any
4	deny tcp any any
5	deny udp 10.1.1.0/26 any
6	deny udp 10.1.1.64/26 any
7	deny udp 10.1.1.128/26 any
8	deny udp 10.1.1.192/26 any
9	deny udp any

Table 2.2: Single ACL with Redundant Rules

of such anomalies is also difficult. Even for a network of moderate size, the number of possible paths between two nodes can be large.

2.3 Rule-Set Redundancy

A firewall needs to inspect a huge number of packets. Therefore, it is difficult not to be concerned with firewall efficiency. A lot of work has been dedicated to improving firewall speed through better hardware and software designs and implementations. To administrators, the most practical way to improve firewall efficiency is through better configuration of the firewall. An efficient firewall configuration should require the minimum possible number of rules, use the least possible amount of memory, and incur the least possible amount of computational load while achieving the filtering goals. Although inefficiency does not directly expose a vulnerability, a faster and more efficient firewall will encourage firewall deployment and therefore make the network safer. In addition, the efficiency of a firewall can determine a network's responsiveness to Denial-of-Service (DoS) attacks.

Redundancy refers to the case where if a rule is removed, the firewall does not change its action on any packets. Reducing redundancy can reduce the total number of rules, and consequently reduce memory consumption and packet classification time [12].

A rule can be considered redundant if the preceding rules have matched a superset of it and specified the same action. For example, in Table 2.2, rule 3 is redundant because rule 2 has already specified the same action for all packets that match rule 3. A rule can also be made redundant by

the subsequent rules. Rules 5, 6, 7 and 8 are all redundant because if we remove them, the packets they would have denied are still going to be denied by rule 9. In fact, for firewalls with a "deny all" policy implicitly appended to the end of each ACL, we do not need rules 4 through 9 at all.

Redundant accept or deny rules are "errors" within the same firewall. However, that is not the case for distributed firewalls. A packet must be accepted on all the firewalls on its path to reach its destination. Redundant accept rules on different firewalls are usually necessary. Redundant deny rules on different firewalls are unnecessary, but are often considered good practice to enhance security, as this redundancy provides an additional line of defense if the outer perimeter is compromised.

CHAPTER 3

RELATED WORK

The complexity of implementing an access control policy through configuration of a large number of distributed devices and the risk of conflicts among these devices have spawned a significant amount of work. We can divide the prior art into several categories.

3.1 Analyzing Rule-Sets for Consistency

These efforts focused on analyzing an existing configuration of one or more access control devices for internal consistency.

3.1.1 Single Firewall Consistency

The majority of work in this area has focused on internal consistency among the rules of a single firewall [13, 14, 15, 16, 17, 18]. Most of this work used variations of rectangular set operations for fast analysis. However, as noted, these approaches are limited to the rule-sets of a single firewall, have no formal error model for the issues that can exist in rule-sets, and do not attempt to check for conformance with any sort of global policy specification.

3.1.2 Inter-Firewall Consistency Analysis

A conflict detection tool for distributed firewall systems has been described in [19], but it only checks for certain kinds of syntactic errors, such as overlapping rules, and not for semantic errors with respect to a specification of the intent. The Fireman tool, proposed in [20], uses binary decision diagrams (BDDs) to check for misconfigurations and inconsistencies in one or more firewalls.

The tool is query-based, being able to indicate only whether a path about which the user has specifically inquired is allowed by the rule-sets. It is unable to provide a complete list of violations of a user-specified global policy. Xie et al. [21, 22, 23, 24] have outlined a limited fault model for firewall rule-set inconsistencies and described ways to automatically suggest corrections when errors are found. Again, they do not identify violations of a user-specified global policy. Furthermore, none of the work takes into account the collective fine-grained access control provided by the network-based mechanisms (e.g., firewalls) and host-based mechanisms (e.g., SELinux policies). Also, none of the above work has been applied to and optimized for specific classes of networks.

Those efforts did not answer the need for ways to model and analyze distributed networks of firewalls, checking for both internal consistency and compliance with policy specification.

3.2 Top-Down Management

Another vector of research effort has focused on automatic generation of consistent policy implementation (at devices) from formal descriptions of global policy [25, 26]. Cisco provides CiscoWorks [27], a suite of products that enable top-down firewall policy management that includes configuration management, a policy manager that generates policies from high-level specifications, and log/audit analysis. Those are appropriate for new, vendor-homogeneous systems, but do not address existing implementations or heterogeneous systems. Furthermore, they require specification of detailed and exhaustive global security policy. For most current network system administrators, the ability to check existing configurations against policy specifications that indicate intended high-level behavior would likely be more useful.

Those efforts did not fill the need for mechanisms that support modeling and analysis of a wide variety of sources for access control rule-sets.

3.3 Attack Graphs

The problem of checking implementation against specification is also known as “model checking” [28]. This line of research has a rich history in the context of proving the correctness of both

hardware and software. The key abstraction is a finite-state machine, and the key problem is that the size of the state space explodes with the complexity of the system. As we consider integration of higher layers of access control mechanisms, the finite state machine view may be appropriate for them. However, while it is possible to map *network* access control into a classical model-checking framework, we aren't convinced of the value of doing so. Much of the attention in model-checking is paid to compact representation of the state space; we already have a representation of the problem (i.e., the rule-graph, to be described), which is a graph whose number of nodes is linear in the number of rules. Furthermore, our analysis takes advantage of the problem domain rather than a general state space, with an approach that lends itself naturally to optimizations for scalability.

3.3.1 Without Statistical Analysis

The research on "attack graphs" is of some relevance to the scalability issue, particularly those research efforts are compared to the statistical analysis that we have developed. Those research efforts have largely been an application of model-checking. Ritchey and Ammann [29] use model checking to identify a *single* violating path in an attack graph. Sheyner et al. [30] provide a more comprehensive analysis, but it suffers from the state-space explosion problem, since the entire attack graph needs to be analyzed to provide the relevant metrics, thereby severely limiting the scalability.

3.3.2 With Statistical Analysis

Ou et al. [31, 32] have used a prolog-based approach, rather than the traditional model-checking, for attack graph generation and analysis; however, their solution does not seem to be able to scale to large networks that are also deep (i.e., graphs with potentially long paths), or to calculate generic metrics that are functions of paths rather than edges in the graph. Our approach includes a statistical analysis component for improved scalability when analyzing large networks, which builds and expands on the mathematical techniques we first developed for model-based penetration testing [33].

3.4 Policy Specification and Composition

As we have indicated, a comprehensive mechanism for analyzing access control policy implementations would include checking for compliance against a user-specified high-level policy that describes the intended behavior. Hence, there's a need to choose or develop a means for writing the said global policy specification.

3.4.1 Authorization Policy Specification Languages

There is considerable academic and industrial interest in XACML (eXtensible Access Control Markup Language) [34], a core XML schema for representing authorization and entitlement policies, which is an open standard developed and promoted by the Oasis XACML Technical Committee. Other researchers are exploring problems such as mapping into XACML of simple and intuitive global policy languages for which policies have been verified, combining XACML and SAML (Security Assertion Markup Language) to support distributed authorization [35], and analysis of the impact on the global policy stance of (even small) changes in an XACML policy.

RCL2000 [36] is a specification language for role-based authorization constraints. It can be used to express prohibition and obligation constraints. Ponder [37] is a language for specifying management and security policies for distributed systems.

All of the above specification languages are similar in expressive power and largely interconvertible. They have the added benefit of being human-readable. However, the descriptions involved can be highly verbose, and any formal reasoning on the specifications can be difficult and inefficient.

3.4.2 Logics

These include first-order predicate logic, and the temporal logics: Linear Temporal Logic (LTL) [38] and Computation Tree Logic (CTL) [39]. They are widely used as the specification languages for various model-checking tools. The specifications in these logics can be compact, lend themselves to efficient evaluation, and facilitate analytical reasoning. However, they can be difficult

for non-experts to write and comprehend. Furthermore, while first-order predicate logic may be insufficient, full temporal logics may be an overkill for most access control policies.

The above literature on specification formalisms did not fill the need for a formalism that can be used to write human-readable specifications but still provide a clear mapping to a formal mathematical system.

3.5 Other Commercial Tools

Vendor-neutral tools have recently appeared that consider access control in a distributed system. They include the Skybox firewall compliance auditor [40] and Red Seal SRM [41]. Both tools use network topology and routing information to determine potential network flows. Red Seal SRM uses vulnerability databases and specification of software running on hosts to compute risk measures, while Skybox determines when firewall rules allow violation of a more abstractly stated global policy. In comparison, our approach incorporates sophisticated statistical analysis for highly improved scalability and also naturally admits integration of higher-level layers of access control policy (e.g., SELinux policies, and/or role-based access in trusted networks).

Among other related commercial products are Netflow and Peakflow [42] (from Arbor Networks), which are tools for network management and run-time anomaly detection, and focus on alert aggregation, correlation, and traffic visualization. They are event-driven and do not detect problems with the security configuration itself. That is, they perform failure detection as opposed to fault detection as they are able to detect vulnerabilities resulting from misconfigurations only when those vulnerabilities result in actual exploits.

CHAPTER 4

ANALYSIS FRAMEWORK

4.1 Analysis Framework: Formalisms

In this section, we introduce a formal framework for reasoning about and evaluating the conformance of access control implementation with a specification of the policy in networked systems.

4.1.1 Formalisms for Global Access Policy

Our formalism for specifying the global access policy is an XML-based specification language inspired by XACML. The XML schema for the language is shown in Figure 4.1. We will describe formalisms as applied to network layer access control.

Definition 1. *A network traffic terminus is a tuple* (A, P, S).

- *A is a set of disjoint IP addresses* $\{a_1, a_2, \cdots, a_n\}$.
- *P is a (layer 3) network protocol.*
- *S is a set of port numbers or other protocol-specific identifiers (e.g., ICMP message type if* $P =$ `ICMP`*),* $\{s_1, s_2, \cdots, s_n\}$.

$\mathscr{V}$ is the set of all possible network traffic terminuses.

Definition 2. *A constraint is used to indicate how a particular traffic flow is to be treated. It is defined as the tuple* $\psi = (V^s, V^d, \lambda, \omega)$.

- *The first two parameters* ($V^s, V^d \in \mathscr{V}$) *are the source and destination terminuses respectively.*
- $\lambda \in \{auth, nonauth, any\}$ *indicates if the constraint is applicable to authenticated traffic.*

```
<?xml version='1.0' encoding='us-ascii'?>

<!DOCTYPE PolicySet [

<!ELEMENT PolicySet (NetworkGroup*, ServiceGroup*, ICMPGroup*,
                     ProtocolGroup*, Policy*)>

<!ELEMENT NetworkGroup (Description, AddressBlock+)>
<!ATTLIST NetworkGroup name CDATA #REQUIRED>
<!ELEMENT AddressBlock EMPTY>
<!ATTLIST AddressBlock NetAddress CDATA #REQUIRED>
<!ATTLIST AddressBlock NetMask CDATA #REQUIRED>

<!ELEMENT ServiceGroup (Description, PortRange+)>
<!ATTLIST ServiceGroup name CDATA #REQUIRED>
<!ELEMENT PortRange EMPTY>
<!ATTLIST PortRange protocol CDATA #REQUIRED>
<!ATTLIST PortRange beginPort CDATA #REQUIRED>
<!ATTLIST PortRange endPort CDATA #REQUIRED>

<!ELEMENT ICMPGroup (Description, ICMPType+)>
<!ATTLIST ICMPGroup name CDATA #REQUIRED>
<!ELEMENT ICMPType EMPTY>
<!ATTLIST ICMPType type CDATA #REQUIRED>

<!ELEMENT ProtocolGroup (Description, ProtocolID+)>
<!ATTLIST ProtocolGroup name CDATA #REQUIRED>
<!ELEMENT ProtocolID EMPTY>
<!ATTLIST ProtocolID protocol CDATA #REQUIRED>

<!ELEMENT Policy (Constraint*)>
<!ATTLIST Policy name CDATA #REQUIRED>
<!ELEMENT Constraint EMPTY>
<!ATTLIST Constraint name CDATA #REQUIRED>
<!ATTLIST Constraint nodeRef CDATA #REQUIRED>
<!ATTLIST Constraint type CDATA #REQUIRED>
<!ATTLIST Constraint direction CDATA #REQUIRED>
<!ATTLIST Constraint sourceIP CDATA #REQUIRED>
<!ATTLIST Constraint sourceMask CDATA #REQUIRED>
<!ATTLIST Constraint sourcePortRange CDATA #REQUIRED>
<!ATTLIST Constraint destinationPortRange CDATA #REQUIRED>
<!ATTLIST Constraint ipProtocol CDATA #REQUIRED>
<!ATTLIST Constraint description CDATA #REQUIRED>
<!ATTLIST Constraint authenticatedTraffic (true | false) #REQUIRED>
<!ATTLIST Constraint users CDATA 'N/A'>
<!ATTLIST Constraint roles CDATA 'N/A'>
<!ATTLIST Constraint domains CDATA 'N/A'>

<!ELEMENT Description (#PCDATA)>
]>
```

Figure 4.1: Global Access Policy Document Type Definition (DTD)

- ω *is the type or action of the constraint. It can take a value from the set* $\{allow_{exact}$, $allow_{min}$, $deny\}$.
  - $allow_{exact}$ *indicates that a path* must *exist between the two terminuses.*
  - $allow_{min}$ *indicates that a path* can *exist between the two terminuses. This type of constraint is used to refine broader deny constraints, to facilitate compactness of specification. This would become clearer as we describe the* implicit deny constraint *and the* principle of greatest specificity *below.*
  - $deny$ *indicates that a path* must not *exist between the two terminuses.*

$\Psi_{explicit} = \{\psi_1, \psi_2, \cdots, \psi_n\}$ is the set of all explicitly defined constraints.

Definition 3. *An implicit deny constraint exists for each policy, and is defined to be* $\psi_{implicit}$. $\psi_{implicit} = ({V^s}_{allow}, {V^d}_{allow}, any, deny)$.

- ${V^s}_{allow} = \bigcup_{(V^s,V^d,\lambda,\omega)\in\Psi_{explicit},s.t.\omega\in\{allow_{exact},allow_{min}\}} V^s$. *We define the union of the terminuses as the terminus formed by the union of the corresponding elements (sets themselves) in their tuples.*
- ${V^s}_{deny} = \bigcup_{(V^s,V^d,\lambda,\omega)\in\Psi_{explicit},s.t.\omega\in\{allow_{exact},allow_{min}\}} V^d$. *We define the union of the terminuses as the terminus formed by the union of the corresponding elements (sets themselves) in their tuples.*

The constraint $\psi_{implicit}$ indicates that unless *refined* by other constraints in the policy, all traffic between any two terminuses mentioned in the other *allow*-type constraints is denied. This allows us to define compact policy specifications using just a collection of (mostly) $allow_{min}$ constraints to describe accepted (but not necessary) behavior, denying everything else.

Definition 4. *A conditionally composed constraint,* $\psi = \psi_1|\psi_2$ *is composed of the constraints* ψ_1 *and* ψ_2. *It indicates that the constraint* ψ_1 *is applicable only if the constraint* ψ_2 *is true for a traffic flow. The constraint is considered to be met if* ψ_2 *is not true.*

Conditionally composed constraints are useful for modeling stateful firewalls, flows with multiple sessions and encapsulated traffic.

Definition 5. *A global access policy specification,* Π*, is an ordered pair* (Σ, Ψ).

- Σ *is a set of variable definitions for various network and traffic elements, including IP address blocks, network protocols, and services (ports).*
- Ψ *(*$= \{\psi_1, \psi_2, \cdots, \psi_n, \psi_{implicit}\}$*) is a set of policy constraints defined on the elements of* Σ.

It is possible for a flow to have multiple rules apply to it, and have conflicts in the rule outcomes. Coupled with a good conflict resolution mechanism, that is actually quite a useful tool for compact specification of desired policy. For example, imagine that an installation wants to follow the best-practice recommendation that has traffic from the control network and from the corporate network terminate in the DMZ, except for one specific application for which a host named *control-database* in the control network must be able to upload data to a host named *corporate-database* in the corporate network. That goal would be accomplished through the creation of two constraints. The first would implement the recommendation as we have already described; this is a "deny" rule. The second would be a "permit" rule that names *control-database* and the specific port used as the source set, names *corporate-database* and the specific port used as the destination set, binds the constraint to control-database, makes the direction "outgoing," and names the protocol (e.g., TCP) used to carry the connection. The union of these two rules does what we want, so long as the conflict resolution mechanisms applies the second rule to the specific flow of interest, and applies the first rule to all other flows coming out of the control network.

We resolve rule conflicts using the *principle of greatest specificity*. This principle, well-known to system administrators who manage routers, is that if there are multiple forwarding rules that apply to the destination of an incoming packet, the rule with the largest number of significant prefix bits ("longest prefix") is the one chosen. Applied to our context, a rule that in all ways is more narrowly applicable to sources, destinations, and protocols than another ought to take precedence over it, because it is more specific in all of its descriptive details. Clearly that is exactly the case in the example described in the previous paragraph.

Definition 6. *Let* $V = (A, P, S)$ *and* $V' = (A', P', S')$ *be two network traffic terminuses, such that* $V, V' \in \mathscr{V}$*. We say,*

- $V \subseteq V'$ *if* $A \subseteq A'$, $P \subseteq P'$ *and* $S \subseteq S'$*;*

- *$V \subset V'$ if at least one of the above inclusions is strict.*

The principle of greatest specificity is stated as follows.

Definition 7. *Let $\psi_1 = (V_s, V_d, \lambda, \omega)$ and $\psi_2 = (V_s', V_d', \lambda', \omega')$ be two global policy constraints, where $V_s, V_d, V_s', V_d' \in \mathscr{V}$, $\lambda, \lambda' \in \Lambda$, and $\omega, \omega' \in \Omega$. ψ_2 is more specific than ψ_1 if $V_s \subset V_s'$, $V_d \subset V_d'$ and $\lambda = \lambda'$. In such a case, for any traffic flow to which both ψ_1 and ψ_2 apply, action ω' is taken.*

The principle of greatest specificity implies that if there is a flow for which multiple constraints apply, and among these constraints there is one (say ψ) that is more specific in all particulars than every other constraint in that policy set, then the conflicts are resolved and constraint ψ is applied. The principle of greatest specificity will not resolve all conflicts; if a global policy has unresolved conflicts then the user needs to know what those conflicts are and the characteristics of the traffic flows that are governed by conflicting constraints. The algorithm in Figure 4.2 performs that consistency check, allowing one to define a consistent set of constraints prior to performing an analysis that checks the implementation against the policy.

4.1.2 Formalisms for Access Control Elements

Our goal is to analyze complex networked systems with multiple networks, each containing one or more hosts, connected via various network layer 3 devices (routers/switches), where the access is controlled through firewalls (or similar devices). We also take into consideration any virtual overlays that may be present, such as Virtual Private Networks (VPNs). Our model of the network topology that can be analyzed is represented by the XML schema shown in Figure 4.3. To briefly summarize the representation, all the hosts, networks, access control devices (firewalls), layer 3 routing devices (switches), and overlays (tunnels) are enumerated. Each host stores references to the network(s) to which it belongs; each firewall or switch stores references to the networks it bridges (and information on how it bridges them); and each tunnel stores references to its end points. Some other, more specific, details are stored as well, as shown in the figure.

Firewalls from different vendors may vary significantly in terms of configuration languages, rule organizations, and interaction between lists or chains. However, a firewall generally consists of a few interfaces and can be configured with several access control lists (ACLs). Both the ingress

procedure resolve-constraint-conflict
  $\Psi \leftarrow \{\psi_1, \cdots, \psi_n\}$ # *set of constraints in the policy*
  $C \leftarrow \emptyset$ # *set of conflicting constraints*
  for all $\psi_i \in \{\psi_1, \cdots, \psi_{n-1}\}$ **do**
    for all $\psi_j \in \{\psi_{i+1}, \cdots, \psi_n\}$ **do**
      $(V^s{}_i, V^d{}_i, \lambda_i, \omega_i) \leftarrow \psi_i$; $(V^s{}_j, V^d{}_j, \lambda_j, \omega_j) \leftarrow \psi_j$
      if $V^s{}_i \cap V^s{}_j \neq \emptyset$ and $V^d{}_i \cap V^d{}_j \neq \emptyset$ and $\omega_i \neq \omega_j$ **then**
        $resolution_1 \leftarrow$ subset $(V^s{}_i, V^s{}_j)+$ subset $(V^d{}_i, V^d{}_j)$
        # *subset*(x,y) *returns* 1 *if* $x \subset y$, 2 *if* $x = y$, *and* 0 *otherwise*
        $resolution_2 \leftarrow 0$
        if $resolution_1 \leq 1$ or $resolution_1 \geq 4$ **then**
          $resolution_2 \leftarrow$ subset $(V^s{}_i, V^s{}_j)+$ subset $(V^d{}_i, V^d{}_j)$
          if $resolution_1 \leq 1$ or $resolution_1 \geq 4$ **then**
            $C \leftarrow C \cup (\psi_i, \psi_j)$
          end if
        end if
      end if
    end for
  end for
  if $C \neq \emptyset$ **then**
    return C
  end if
end procedure

Figure 4.2: Algorithm for Resolving Policy Constraints

and egress of an interface can be associated with ACLs. If an ACL is associated with the ingress, filtering is performed when packets arrive at the interface. Similarly, if an ACL is associated with the egress, filtering will be performed before packets leave the interface. Each ACL consists of a list of rules. Individual rules can be interpreted in the form $< P, action >$, where P is a predicate describes what packets are matched by this rule and *action* describing the corresponding action performed on the matched packets. Packets not matched by the current rule will be forwarded to the next rule until a match is found or the end of the ACL is reached. At the end of an ACL, the default action will be applied. This is similar to an "if-elsif-else" construct in most programming languages. Implicit rules vary on different firewall products. On the Cisco PIX firewall and routers, the implicit rule at the end of an ACL denies everything. On Linux Netfilter, the implicit rule is defined by the policy of the chain.

Note that so far, we have described the so-called "first-matching" ACLs. Some firewalls, e.g.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE Topology [
<!ELEMENT Topology (Network*, Host*, Switch*, Firewall*, Tunnel*)>
<!ATTLIST Topology isEncrypted (true|false) #REQUIRED>

<!ELEMENT Network (Description)>
<!ATTLIST Network name CDATA #REQUIRED>
<!ATTLIST Network isCompletelyConnected (true|false) #REQUIRED>
<!ATTLIST Network netAddress CDATA #REQUIRED>
<!ATTLIST Network netMask CDATA #REQUIRED>

<!ELEMENT Host (Description, IPAddress+, NodeRef*)>
<!ATTLIST Host name CDATA #REQUIRED>
<!ATTLIST Host accessParameters CDATA #REQUIRED>

<!ELEMENT Switch (Description, IPAddress, NodeRef*)>
<!ATTLIST Switch name CDATA #REQUIRED>

<!ELEMENT Firewall (Description, IPAddress+, NodeRef*)>
<!ATTLIST Firewall name CDATA #REQUIRED>
<!ATTLIST Firewall accessParameters CDATA #REQUIRED>
<!ATTLIST Firewall NAT CDATA #REQUIRED>

<!ELEMENT NodeRef EMPTY>
<!ATTLIST NodeRef name CDATA #REQUIRED>

<!ELEMENT IPAddress EMPTY>
<!ATTLIST IPAddress ipAddress CDATA #REQUIRED>

<!ELEMENT Tunnel (Description, IPAddress+, NodeRef+)>
<!ATTLIST Tunnel encryption CDATA #IMPLIED>

<!ELEMENT Description (#PCDATA)>
]>
```

Figure 4.3: Network Topology DTD

```
<?xml version='1.0' encoding='us-ascii'?>

  <!DOCTYPE RuleSetCollection [

  <!ELEMENT RuleSetCollection (RuleSet*)>

  <!ELEMENT RuleSet (Description, IPInterface*, NetworkGroup*,
                     ServiceGroup*, ICMPGroup*, ProtocolGroup*,
                     AccessList*)>
  <!ATTLIST RuleSet name CDATA #REQUIRED>
  <!ATTLIST RuleSet fragsAllowed (true | false) #REQUIRED>
  <!ATTLIST RuleSet nonIPTrafficAllowed (true | false) #REQUIRED>
  <!ATTLIST RuleSet sniffingAllowed (true | false) #REQUIRED>
  <!ATTLIST RuleSet spoofingAllowed (true | false) #REQUIRED>
  <!ATTLIST RuleSet ipOptionAllowed (true | false) #REQUIRED>
  <!ATTLIST RuleSet testMode (true | false) #REQUIRED>
  <!ATTLIST RuleSet fallbackMode CDATA #REQUIRED>

  <!ELEMENT IPInterface EMPTY>
  <!ATTLIST IPInterface name CDATA #REQUIRED>
  <!ATTLIST IPInterface address CDATA #REQUIRED>
  <!ATTLIST IPInterface netmask CDATA #REQUIRED>

  <!ELEMENT NetworkGroup (Description, AddressBlock+)>
  <!ATTLIST NetworkGroup name CDATA #REQUIRED>
  <!ELEMENT AddressBlock EMPTY>
  <!ATTLIST AddressBlock NetAddress CDATA #REQUIRED>
  <!ATTLIST AddressBlock NetMask CDATA #REQUIRED>

  <!ELEMENT ServiceGroup (Description, PortRange+)>
  <!ATTLIST ServiceGroup name CDATA #REQUIRED>
  <!ELEMENT PortRange EMPTY>
  <!ATTLIST PortRange protocol CDATA #REQUIRED>
  <!ATTLIST PortRange beginPort CDATA #REQUIRED>
  <!ATTLIST PortRange endPort CDATA #REQUIRED>

  <!ELEMENT ICMPGroup (Description, ICMPType+)>
  <!ATTLIST ICMPGroup name CDATA #REQUIRED>
  <!ELEMENT ICMPType EMPTY>
  <!ATTLIST ICMPType type CDATA #REQUIRED>

  <!ELEMENT ProtocolGroup (Description, ProtocolID+)>
  <!ATTLIST ProtocolGroup name CDATA #REQUIRED>
  <!ELEMENT ProtocolID EMPTY>
  <!ATTLIST ProtocolID protocol CDATA #REQUIRED>
```

Figure 4.4: Access Control Elements DTD: Part 1

```
<!ELEMENT AccessList (Description, AuthorizedUsers*, Rule*)>
<!ATTLIST AccessList name CDATA #REQUIRED>
<!ATTLIST AccessList incomingInterface CDATA #REQUIRED>
<!ATTLIST AccessList authenticationACL (true | false) #REQUIRED>

<!ELEMENT AuthorizedUsers (User*)>
<!ATTLIST AuthorizedUsers aaaServerAddress CDATA #REQUIRED>
<!ATTLIST AuthorizedUsers aaaServerProtocol CDATA #REQUIRED>

<!ELEMENT User EMPTY>
<!ATTLIST User name CDATA #REQUIRED>

<!ELEMENT Rule      (Description)>
<!ATTLIST Rule name CDATA #REQUIRED>
<!ATTLIST Rule sourceHostID CDATA #REQUIRED>
<!ATTLIST Rule sourcePortRange CDATA #REQUIRED>
<!ATTLIST Rule sourceMask CDATA #REQUIRED>
<!ATTLIST Rule destinationHostID CDATA #REQUIRED>
<!ATTLIST Rule destinationPortRange CDATA #REQUIRED>
<!ATTLIST Rule destinationMask CDATA #REQUIRED>
<!ATTLIST Rule direction CDATA #REQUIRED>
<!ATTLIST Rule action CDATA #REQUIRED>
<!ATTLIST Rule ipProtocol CDATA #REQUIRED>
<!ATTLIST Rule enabled (true | false) #REQUIRED>
<!ATTLIST Rule audit CDATA #REQUIRED>
<!ATTLIST Rule testMode (true | false) #REQUIRED>
<!ATTLIST Rule ruleNegated (true | false) #REQUIRED>
<!ATTLIST Rule allowTcpConnectInit (true | false) #REQUIRED>

<!ELEMENT Description (#PCDATA)>
]>
```

Figure 4.5: Access Control Elements DTD: Part 2

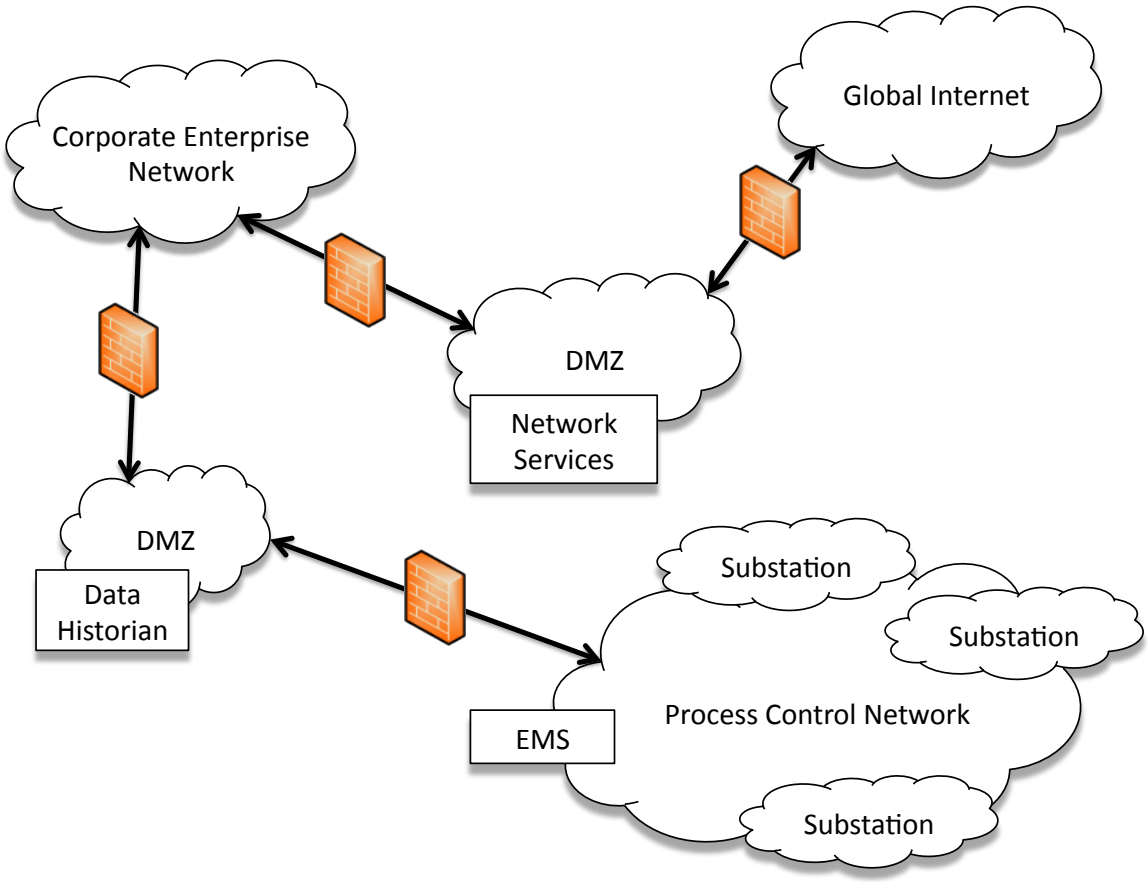

Figure 4.6: An Example Process Control Network

BSD Packet Filter, use "last-matching" ACLs, in which the decision applied to a packet is determined by the last matched rule. An ACL using last-matching can be converted to a first-matching form via re-ordering. In this dissertation, we assume that every ACL uses first-matching. Traditional stateless firewalls treat each packet in isolation and check every packet against the ACL; this is computation-intensive and often a performance bottleneck. Modern stateful firewalls can monitor TCP three-way handshakes and build an entry in the state table. If the firewall matches a packet to an *established* flow, it can accept it without checking the ACL, thus significantly reduce the computation overhead. However, the ACLs still determine whether a state can be established in the first place. Therefore, the correct configuration of ACLs is important even for stateful firewalls.

Depending on the available "actions" and rule execution logic, we classify firewalls into two typical models: (1) the simple list model, which is represented by Cisco PIX firewall and router ACLs, and (2) the complex chain model, which is represented by Linux Netfilter. Firewalls using

the simple list model allow only accept and deny actions. The complex chain model, in addition to "accept" and "deny," also supports calling of another user-defined chain or "return." We use rule graphs to model the control flow of ACLs. As can be seen in Section 4.2.2, the rule graph of ACLs that use the simple list model is just the access control list itself. The rule graphs for ACLs using the complex chain model are similar to control-flow graphs in programming languages.

In a typical network environment, multiple firewalls are often deployed across the network in a distributed fashion. Although firewalls are configured independently, a network depends on the correct configuration of all related firewalls to achieve the desired end-to-end security behavior. By "end-to-end security behavior," we refer to the decision on whether a packet should be allowed to reach a protected network. The packet can be going from one side of a Virtual Private Network (VPN) to another side of the VPN. It can be going from the untrusted Internet to the trusted secured intranet.

See Figure 4.6, for example. An enterprise network is connected to the Internet, and the top two firewalls in the figure are deployed to guard the first Demilitarized Zone (DMZ). Services such as Web and email that must allow public access are more vulnerable (and hence less trustworthy) and are normally put in the DMZ. Further inside, the internal process control network is guarded by two additional firewalls (with another DMZ in between). In general, that pair of firewalls will have a tighter security policy. Important applications and sensitive data often run inside the internal trusted networks, and only limited accesses are allowed. Since there are multiple paths from the Internet to the internal network, the filtering action taken depends on the path a packet actually traverses. Although a packet does not actually choose its data path, the dynamics of the underlying routing plane may assign different paths for the same set of packets at different times. Ideally, firewalls should perform consistently regardless of the underlying routing decisions. To guarantee reachability of desired packets, the administrator must ensure that none of the firewalls on the path deny them. On the other hand, the administrator must ensure that no potential path allows prohibited packets to access the protected network.

Definition 8. *A rule in an ACL is one of the following,*

- $< P, accept >$*: accept the packet*
- $< P, deny >$*: deny the packet*

- $< P, chainY >$: *goto ACL chain "Y"*
- $< P, return >$: *resume calling chain*

where P indicates the predicate characterizing the attributes of the traffic being matched against the rule.

We have developed an XML-based unified schema, shown in Figure 4.4 and Figure 4.5, that provides an intermediate representation for several types and makes of firewalls. We can currently model rule-sets from Cisco PIX and ASA firewalls, Checkpoint firewalls, SonicWall firewalls, and Linux iptables. We model an extensive list of features and functionality, including:

- Static and dynamic network address translation (NAT).
- Static routing.
- Authentication, Authorization and Auditing (AAA).
- Virtual Private Networks (VPNs).
- ACL chains.
- Virtual addresses.

4.2 Analysis Framework: Data Structures

4.2.1 Multidimensional Interval Tree

We use *multidimensional interval trees* [43, 44] to represent the predicate characterizing the traffic that a particular rule matches. First, we establish a mapping between traffic attribute types and interval tree dimensions.

Definition 9. *A traffic attribute predicate can be characterized as a tuple* $(\mathscr{S}, \mathscr{D}, \mathscr{Q}, \mathscr{P})$*, where* $\mathscr{S}$ *is a range of source IP addresses,* $\mathscr{D}$ *is a range of destination IP addresses,* $\mathscr{Q}$ *is a layer 3 protocol, and* $\mathscr{P}$ *is a range of network port numbers. The mapping is as follows.*

- $\mathscr{S} \leftrightarrow [x^1{}_{min}, x^1{}_{max}]$, $x^1{}_{min}$ = *minimum address in range,* $x^1{}_{max}$ = *maximum address in range*
- $\mathscr{D} \leftrightarrow [x^2{}_{min}, x^2{}_{max}]$, $x^2{}_{min}$ = *minimum address in range,* $x^2{}_{max}$ = *maximum address in range*
- $\mathscr{Q} \leftrightarrow [x^3{}_{min}, x^3{}_{max}]$, $x^3{}_{min} = x^3{}_{max}$ = *protocol number*
- $\mathscr{P} \leftrightarrow [x^4{}_{min}, x^4{}_{max}]$, $x^4{}_{min}$ = *minimum port number in range,* $x^4{}_{max}$ = *maximum port number in range*
- *Any* $\leftrightarrow [x^i{}_{min}, x^i{}_{max}]$, $x^i{}_{min} = D^i{}_{min}$, $x^i{}_{max} = D^i{}_{max}$; $D^i{}_{min}$ *and* $D^i{}_{max}$ *are the minimum and maximum possible values of dimension i*

Thus, the ACLs we have discussed so far would map to an interval tree node with 4 dimensions. Using that mapping, we can easily convert any rule predicate into a multidimensional interval tree node. An interval tree is a binary tree based on the end points of intervals. For example, let X be a set of intervals in one dimension, and let m be the median of the interval end points. The set of intervals containing m is stored at the root. The set of intervals to the left or right of m forms the left or right subtree. These intervals are recursively partitioned based on their median. Search, insert, and delete operations in a one-dimensional interval tree take $O(\lg n)$ time.

A multidimensional interval tree is a straightforward generalization of the one-dimensional interval tree. An interval tree can be initially constructed based on only the intervals in the first dimension. The resulting first-level interval tree (sometimes referred to as a *component tree*) contains a set of elements in each tree node. Each node in turn stores its element using an interval tree based on the second dimension. Any query is reduced to a sequence of binary searches in each dimension. The multidimensional interval tree data structure implemented in our work is a dynamic interval tree in the sense that it can support inserts and deletes. Query, insert, and delete operations take $O(n \lg^d n)$ time, where d is the number of dimensions.

Similarly to rule predicates, we use multidimensional interval trees to represent network traffic along a path in the network. The *traffic attribute set (TAS)* is the data structure used to represent a collection of packets arriving at any point in the network. Each TAS is a set of 4-tuples, where each tuple is of the form *(source address range, destination address range, protocol number, port range)*. We represent each TAS as a 4-dimensional interval tree, similar to the discussion above.

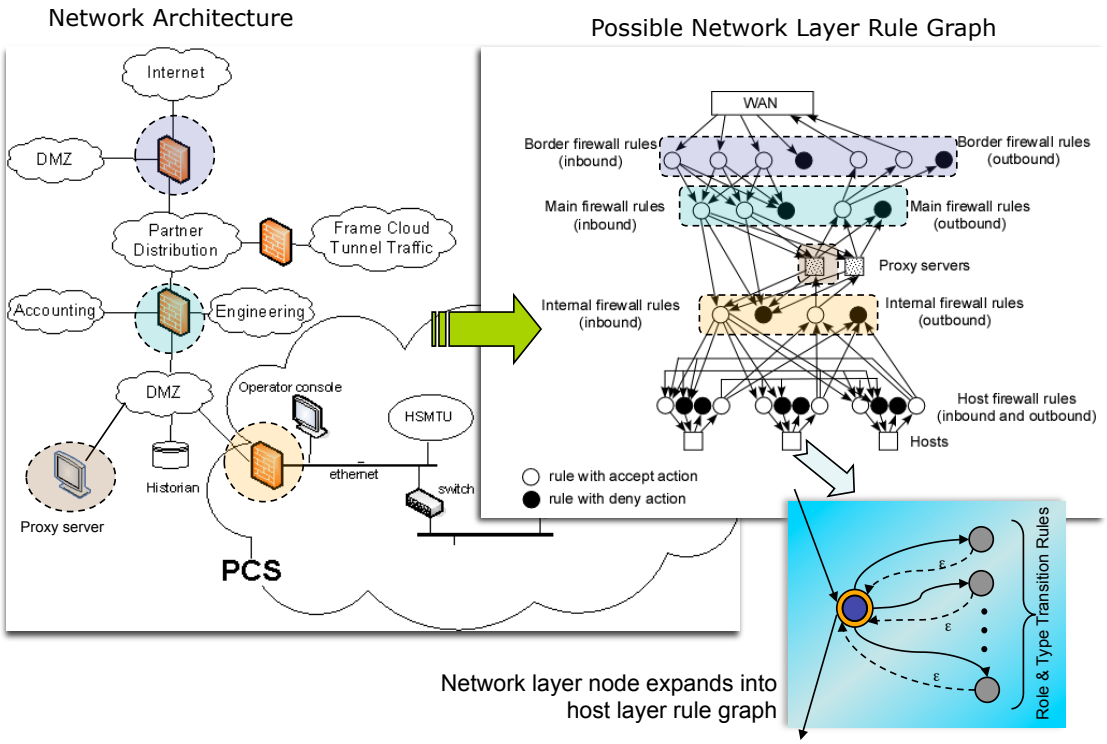

Figure 4.7: Constructing Rule Graphs

In the subsequent sections, whenever we discuss "network traffic" or a "collection of packets," they are assumed to be actually represented by traffic attribute sets.

4.2.2 Multilayered Rule-Graph

The network topology and the configuration information (policy rules) from the access control devices are internally represented by the consistency checker using a specially constructed data structure called a *multilayered rule graph*. Figure 4.7 shows a simplified representation of a possible two-layered rule graph for the network shown on the left side of the figure. The rule graph captures the network interconnectivity and data flow among the policy enforcement rules. In the top-level, or the *network-layer*, rule graph, there are some nodes that correspond to devices that accept traffic and other nodes that correspond to rules in devices through which traffic passes. A firewall (dedicated or host-based) is represented by as many nodes as it has rules, with both inbound and outbound rules being presented; a host (terminal or proxy) is represented by just one node. A node representing rule r_i in firewall F_m directs an arc to a node representing rule r_j in

firewall F_n if 1) the action (access control decision) associated with the rule r_i is to accept, and 2) a network packet that is thus accepted can be received (in accordance with the network topology) by firewall F_n. A node representing a proxy host (that allows for traffic to pass through it) has both incident and outgoing arcs.

Nodes representing hosts may further expand into a lower-level rule graph; Figure 4.7 shows a potential *host-layer rule graph* (modeled as a directed graph with labeled edges) representing the SELinux policy rules for one of the hosts in the network-layer rule graph. The arcs in this rule graph change the security contexts and/or the permissions of various objects present on the host.

The overall strategy used by the consistency checker is to analyze paths from traffic sources to terminal nodes in the rule graph. A terminal node may be a rule associated with a "deny" action, a destination, or an exit from the system. A path describes a possible sequence of access policy decisions applied to all traffic that traverses the corresponding devices in the network. The analysis approach is built around the observation that knowing which rules in the system make decisions about a piece of traffic tells us something about the attributes of that piece of traffic. A network packet arriving from the Internet may in principle carry any source address, any destination address, any protocol, and any set of user privileges. If the packet is accepted by a particular rule in the border firewall, then we can infer that its attributes do not satisfy the preconditions of the earlier rules in the firewall's rule-set, but do satisfy the preconditions of the rule that it matched. Every time a rule recognizes traffic, it refines the attributes of the traffic to reflect the constraining influence of the rules against which the traffic was tested at the firewall. Given a path in the rule graph, the consistency checker can compute successive refinements of the attribute set of the traffic, user classes, and object permissions that can possibly traverse that path. At each stage along a path, the current attribute set is matched against the formal specification of the access constraints to confirm whether it is in violation.

For firewalls using the simple list model, there is no possibility of branching, and the rule graph is the same list. For firewalls using the complex chain model, branching can be caused by calling "chain Y" and "returning" from it. To handle such branching, we introduce $< P, pass >$ to indicate that only packets matching this predicate will remain in this path. For a $< P, chainY >$ rule, we insert $< P, pass >$ before going to "chain Y". We also insert $< \neg P, pass >$ for the path that does not jump to "chain Y."

Recursive function calls should be avoided, since they could create loops. Loops can easily be prevented by ensuring that no rules appear twice on a rule path. Earlier versions of Netfilter denied a packet when it was found to be in a loop. But it is probably better to avoid doing so at configuration time. After loops have been eliminated, the rule graph can be constructed by linearization.

The input to an ACL is denoted by I, which is the collection of packets that can possibly arrive at this access list. For an ACL using the complex chain model, the rule graph may give n rule paths from the input to the output. For each of the n rule paths, we traverse the path to collect information.

For the j^{th} rule $< P_j, action_j >$ in the rule path, we define the current state as $< A_j, D_j, F_j >$, where A_j and D_j denote the network traffic accepted and denied before the j^{th} rule, respectively; F_j denotes the set of packets that have been diverted to other paths. We use R_j to denote the collection of the remaining traffic that can possibly arrive at the j^{th} rule. R_j can be found from the input I and the current state information, as shown in Equation 4.1.

$$R_j = I \cap \neg(A_j \cup D_j \cup F_j) \tag{4.1}$$

For the first rule of an ACL, we have the initial value of $A_1 = D_1 = F_1 = \emptyset$ and $R_1 = I$. After reading each rule, we update the state according to the state transformation defined in Equation 4.2 until the end of each rule path. A state transform "$S_i, r \vdash S_{i+1}$" means that if we read in rule r at state S_i, the result will be state S_{i+1}. Note that R is automatically updated when $< A, D, F >$ changes.

$$\begin{aligned} &< A, D, F >, < P, accept > \vdash < A \cup (R \cap P), D, F > \\ &< A, D, F >, < P, deny > \vdash < A, D \cup (R \cap P), F > \\ &< A, D, F >, < P, pass > \vdash < A, D, F \cup (R \cap \neg P) > \end{aligned} \tag{4.2}$$

At the end of rule path $path_i$, we can identify the packets accepted and denied through this path as A_{path_i} and D_{path_i}, respectively. Since any packet can take only one path, packets accepted by

this ACL are the union of those accepted on all paths, as shown in Equation 4.3. In addition, since the default action of an ACL matches all packets, all packets will be either accepted or denied (Equation 4.4).

$$\begin{cases} A_{ACL} & = \bigcup_{i \in path} A_{path_i} \\ D_{ACL} & = \bigcup_{i \in path} D_{path_i} \end{cases} \tag{4.3}$$

$$\begin{cases} A_{ACL} \cup D_{ACL} & = I_{ACL} \\ R_{ACL} & = \emptyset \end{cases} \tag{4.4}$$

In the network of distributed firewalls, a packet will go through a series of ACLs to reach the destination. It needs to survive the filtering of *all* the ACLs on the path. On the other hand, a well-engineered network often has multiple paths and uses dynamic routing to improve performance and reliability. As a result, a packet could traverse different ACL paths at different times.

Given the topology as a directed graph, one can determine all the possible paths from one node to another. Since each ACL is associated with an individual interface and a direction, one can build a tree of ACLs. Based on information about network connectivity, one can compute the ACL tree rooted at a destination using either Depth-First Search (DFS) or Breadth-First Search (BFS) algorithms. The resulting tree graph reveals all the ACL paths packets may traverse to reach the destination. Note that we choose to be blind about the underlying routing and assume that all the paths that are topologically feasible could be taken. The reason is that routing is designed to be dynamic and adaptive to link failures and loads. In addition, firewall configuration should behave correctly and consistently regardless of the underlying routing dynamics.

For a large and well-connected graph, the number of paths can be large. For the portions of a network that are not involved in packet filtering, and therefore do not interfere with the firewall configurations, we use abstract virtual nodes as representations. This approach can greatly reduce the complexity of the graph but still keep the relevant information. For the network illustrated in Figure 2.1, we use three abstract virtual nodes “outside,” “DMZ,” and “inside,” to indicate the untrusted Internet, DMZ, and trusted internal network, respectively. Data paths between these three virtual nodes are often the primary concern of firewall administrators. Note that this dissertation uses the traffic from “outside” to “inside” as an example for discussion. However, our algorithm is

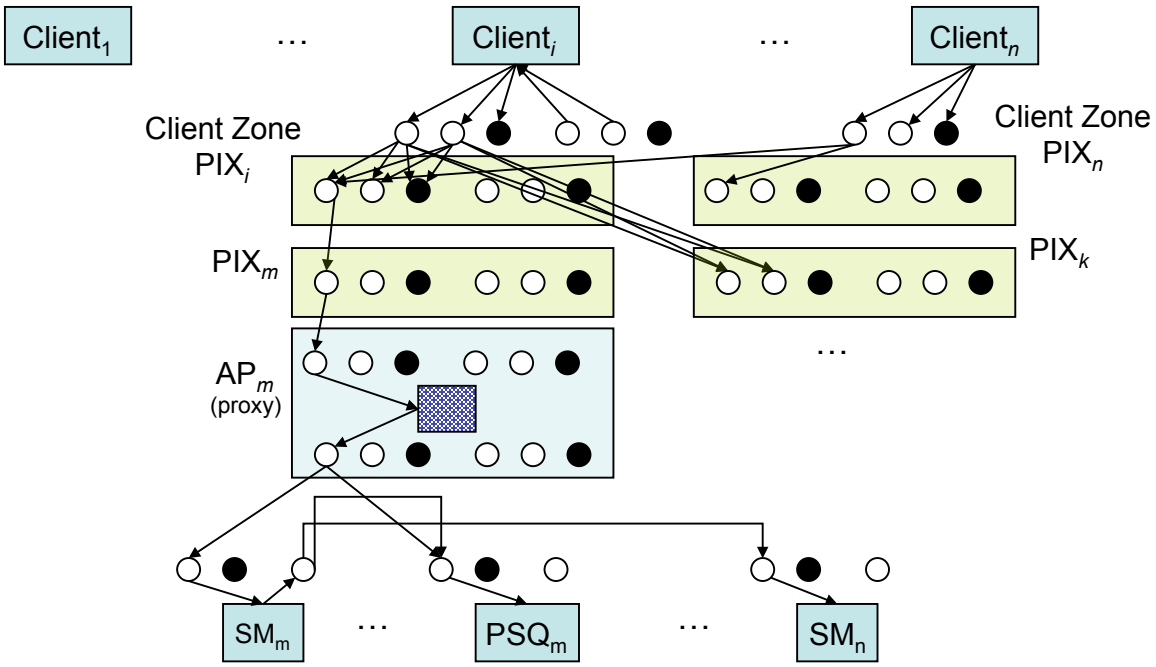

Figure 4.8: A Sample Rulegraph

general enough to consider traffic between any two points in the network.

Figure 4.8 shows the ACL tree built for Figure 2.1. For any given ACL tree graph, ACLs are in series, in parallel, or in a combination of them. For a set of n ACLs in series (or in parallel), packets need to survive the filtering decision of all (or any) of them. Therefore, the accepted set of packets is the intersection (or union) of these ACLs accepted independently. Thus, the final traffic attributes become a sequence of unions and intersections of formulas provided by Equations 4.3 and 4.4.

4.3 Policy Specification Formalism: Comparison with LTL

4.3.1 Brief Background on LTL

Linear temporal logic or *linear-time temporal logic* (LTL) [38] is a modal temporal logic with modalities referring to time. In LTL, one can encode formulas about the future of paths, e.g., a condition will eventually be true, or a condition will be true until another fact becomes true. LTL is a fragment of the more complex CTL*, which also allows for branching of time and quantifiers.

As a result, LTL is sometimes called *propositional temporal logic*.

LTL Syntax

LTL is built up from a finite set of propositional variables *AP*, the logical operators $\neg$ and $\vee$, and the temporal modal operators $\mathscr{X}$ and $\mathscr{U}$.

Definition 10. *Formally, the set of LTL formulas over AP is inductively defined as follows:*

- *if $p \in AP$ then p is an LTL formula;*
- *if ψ and ϕ are LTL formulas then $\neg\psi$, $\phi \vee \psi$, $\mathscr{X}\psi$, and $\phi \mathscr{U} \psi$ are LTL formulas.*

$\mathscr{X}$ is read as "next" and $\mathscr{U}$ is read as "until." Sometimes, $\mathscr{N}$ is used in place of $\mathscr{X}$. In addition to those fundamental operators, there are other logical and temporal operators, defined in terms of the fundamental operators that are used to write LTL formulas succinctly. The additional logical operators are $\wedge$, $\rightarrow$, $\leftrightarrow$, $true$, and $false$.

Proposition 1. *Using equivalences of negation propagation, we can use the operators $\neg$, $\wedge$, $\mathscr{X}$, and $\mathscr{U}$ as the fundamental operators to express any LTL formula.*

Proof. This follows from Definition 10 and negation propagation over disjunction:

$$\neg(\phi \vee \psi) \equiv \neg\phi \wedge \neg\psi$$

□

The additional temporal operators are below.

- $\mathscr{G}$ for always (globally)
- $\mathscr{F}$ for eventually (in the future)
- $\mathscr{R}$ for release
- $\mathscr{W}$ for weakly until

LTL formulas describe the behavior of the variables in AP over a linear series of time steps starting at time zero and extending infinitely into the future. We satisfy such formulas over computations, which are functions that assign truth values to the elements of AP at each time instant. In essence, a computation path π satisfies a temporal formula ϕ if ϕ is true in the zeroth time step of π, π_0.

4.3.2 Why LTL?

LTL is a popular choice for expressing properties in model-checking tools. Computational Tree Logic (CTL) [39] is another candidate, but we feel that LTL makes a better choice for comparison of expressive power. The reasons for this are manifold.

LTL and CTL have very different kinds of logical expressiveness. Since CTL allows for explicit existential quantification over paths, it is more expressive in some cases in which we want to reason about the possibility of the existence of a specific path through the transition system model M, such as when M is best described as a computation tree. For example, there are no LTL equivalents of the CTL formulas $(E\mathscr{X}\,p)$ and $(A\mathscr{G}E\mathscr{F}\,p)$, since LTL cannot express the possibility of p happening on some path (but not necessarily all paths) next time, or in the future. LTL describes executions of the system, not the way in which they can be organized into a branching tree. Intuitively, it is difficult (or impossible) to express in LTL situations in which distinct behaviors occur on distinct branches at the same time. Conversely, it is difficult (or impossible) to express in CTL some situations where the same behavior may occur on distinct branches at distinct times, a circumstance in which the ability of LTL to describe individual paths is quite useful. In fact, situations in which CTL is more suitable are rare, so LTL turns out to be more expressive from a practical point of view than CTL.

From a system design point of view, it is important to be able to write clear and correct specifications to input into the model checker. Usually, we want to describe behavioral, rather than structural, properties of the model, making LTL the better choice for specification, since such properties are easily expressed in LTL but may not be expressible in CTL. For example, we may want to say that p will happen within the next two time steps, $(\mathscr{X}\,p \vee \mathscr{X}\,\mathscr{X}\,p)$, or that if p ever happens, q will happen too, $(\mathscr{F}\,p \rightarrow \mathscr{F}\,q)$, neither of which is expressible in CTL. Similarly, we

cannot state in CTL that if p happens in the future, q will happen in the next time step after that, which is simply $\mathscr{F}(p \wedge \mathscr{X} q)$ in LTL. Worst of all, CTL's nature is such that it takes some effort to determine that these useful properties are not expressible in CTL. Indeed, a thorough comparison of the two logics concludes that CTL is unintuitive, hard to use, ill-suited for compositional reasoning, and fundamentally incompatible with semiformal verification, while LTL suffers from none of these inherent limitations and in general is better suited to model-checking.

LTL is a fair language, whereas CTL is not. That is to say that LTL can express properties of fairness, including both strong fairness and weak fairness, whereas CTL cannot (though CTL model checkers generally allow users to specify fairness statements separately to compensate for this shortcoming). For example, the LTL formula $(\mathscr{G}\mathscr{F} p \rightarrow \mathscr{F} q)$, which expresses the property that infinitely many p's imply an eventual q, is the form of a common fairness statement: a continuous request will eventually be acknowledged. Yet this sentiment is not expressible in CTL. Since fairness properties occur frequently in the specifications we wish to verify about real-life reactive systems, this adds to the desirability of LTL as a specification language.

For another example, the common invariance property $\mathscr{F}\mathscr{G} p$, meaning "at some point, p will hold forever" cannot be expressed in CTL. It is difficult to see why this formula is not equivalent to the CTL formula $\mathscr{A}\mathscr{F}\mathscr{A}\mathscr{G} p$; after all, we are basically claiming that on all paths, there's a point in which p holds in all future states. (The standard semantic interpretation of LTL corresponds to the "for all paths" syntax of CTL. For that reason, we consider there to be an implicit $\mathscr{A}$ operator in front of all LTL formulas when we compare them to CTL formulas.)

Another frequently cited example of the nonintuitiveness of CTL is the fact that the CTL formula $\mathscr{A}\mathscr{X}\mathscr{A}\mathscr{F} p$ is not equivalent to the formula $(\mathscr{A}\mathscr{F}\mathscr{A}\mathscr{X} p)$. Again, the distinction is subtle. The former formula states that, as of the next time step, it is true that p will definitely hold at some point in the future or, in other words, p will hold sometime in the strict future. That formula is equivalent to the LTL formulas $\mathscr{X}\mathscr{F} p$ and $\mathscr{F}\mathscr{X} p$. On the other hand, the meaning of $(\mathscr{A}\mathscr{F}\mathscr{A}\mathscr{X} p)$ is the strictly stronger (and actually quite strange) assertion that on all paths, in the future, there is some point in which p is true in the next time step on all of the branches from that point.

Those examples illustrate why LTL is frequently considered a more straightforward language, better suited to specification and more usable for verification engineers and system designers. LTL is the preferred logic of the two for general property specification and on-the-fly verification. Ver-

ification engineers have found expression of specifications in LTL to be more intuitive, and less error-prone, than in CTL, particularly for verification of concurrent software architectures. The vast majority of CTL specifications used in practice are equivalent to LTL specifications; it is rare that the added ability to reason over computation tree branches is needed, and it frequently requires engineers to look at their designs in an unnatural way. The expressiveness, simplicity, and usability of LTL, particularly for practical applications like compositional verification, or specification debugging, make it a good choice for industrial verification.

4.3.3 Expressing LTL Formulas in Our Formalism

We consider policy specifications to be descriptions of properties of traffic paths through the network. We now show that after an initial set of mappings is established between atomic propositions and constraints, any LTL formula can be translated to our policy specification language. That would prove that our policy specification language is at least as expressive as LTL.

As stated above, LTL formulas are defined over a set of propositional variables, AP. Every member of AP is an LTL formula. For a given network, we consider each $p \in AP$ to represent some traffic flow through the network that is either allowed or denied.

In general, an LTL formula can be satisfied by an infinite sequence of truth evaluations of the propositional variables in AP. Each such sequence can be considered to be an ω-*word* over the alphabet 2^{AP}. We can consider each ω-word to represent a path through the network, or more specifically, a path through the equivalent rule-graph. In view of those semantics attached to the LTL formulas, we restrict our current discussion to finite length paths through the network (i.e., we only consider ω-words of finite length).

Definition 11. *Given a set of propositional variables AP, each member of which represents an allowed or denied path through the rule-graph for the given network, we construct the initial set of constraints as follows:* $\forall \phi_i \in AP$*, define a non-conditional constraint* ψ_i*, where* ψ_i *characterizes the traffic that* ϕ_i *represents and* $\psi_i.\omega = allow_{exact}$ *or deny depending on whether* phi_i *represents allowed or denied traffic. We consider* $\forall \phi_i \in AP, \phi_i \equiv \psi_i$.

Now to show that any LTL formula (given our restrictions) can be expressed in our policy specification language, we will show that the LTL formulas formed through the use of the fundamental

operators shown in Proposition 1 are equivalent to constraints already present in those defined in Definition 11, or can be constructed and added to the set of all constraints.

Proposition 2. *(Negation) If ϕ_i is an LTL formula and ψ_i is its equivalent constraint, then $\neg\phi_i$ is equivalent to the constraint formed by changing $\psi_i.\omega$ from $allow_{min}$ to deny and vice versa. The new constraint is henceforth represented by $\neg\psi_i$.*

Proof. Let $w = a_1, a_2, \ldots, a_n$, where $a_i \in 2^{AP}$, be an ω-word or equivalently a traffic flow through the given network. We know that $w \models \neg\phi_i$ if $w \not\models \phi_i$. Now, if $w \not\models \phi_i$, we know from Definition 11 that $w \not\models \psi_i$. That means the traffic represented by w is denied by psi_i if $psi_i.\omega = allow_{exact}$ or allowed by psi_i if $psi_i.\omega = deny$. Therefore, if we toggle $psi_i.\omega$ (forming $\neg\psi_i$), traffic represented by w would satisfy the new constraint. Hence, $w \models \neg\psi_i$. Consequently, we have shown that

$$w \models \neg\phi_i \Rightarrow w \models \neg\psi_i$$

We can provide a similar argument in the other direction. □

Proposition 3. *(Conjunction) Let ϕ_i and ϕ_j are LTL formulas, and ψ_i and ψ_j are their equivalent constraints.*

- *If ψ_i and ψ_j are of same type (both allow or both deny), then $\phi_i \wedge \phi_j \equiv \psi_i \wedge \psi_j$, where $(\psi_i \wedge \psi_j) = (\psi_i.V^s \cup \psi_j.V^s, \psi_i.V^d \cup \psi_j.V^d, \lambda, \psi_i.\omega)$. $\lambda = \psi_i.\lambda$ if $\psi_i.\lambda = \psi_j.\lambda$, $\lambda = any$ otherwise.*

- *If ψ_i and ψ_j are of different types, say $\psi_i.\omega = allow_{exact}$ and $psi_j.\omega = deny$, then $\phi_i \wedge \phi_j \equiv \psi_i \wedge \psi_j$, where $(\psi_i \wedge \psi_j) = (\psi_i.V^s \cup \neg\psi_j.V^s, \psi_i.V^d \cup \neg\psi_j.V^d, \lambda, \psi_i.\omega)$. $\lambda = \psi_i.\lambda$ if $\psi_i.\lambda = \psi_j.\lambda$, $\lambda = any$ otherwise.*

Proof. This is evident from the construction. Note that the union of two traffic terminuses if the tuple formed by calculating the set union of corresponding elements of each tuple. Similarly, the negation of a traffic terminus is the tuple obtained by calculating the set subtraction from the corresponding universal set for each element of the tuple. □

Proposition 4. *(Next) Let the rule-graph contain m edges. Let $E_1, E_2, \ldots, E_m$ be constraints allowing traffic to travel along each edge in the rule-graph respectively (and nothing else). Let ϕ_i be an LTL formula and ψ_i be its equivalent constraint. Then,*

$$\mathscr{X}\,\phi_i \equiv (\psi_i|E_1) \wedge \ldots \wedge (\psi_i|E_m)$$

.

Proof. Let $w = a_1, a_2, \ldots, a_n$, where $a_i \in 2^{AP}$, be an ω-word or equivalently a traffic flow through the given network. Let $w(i) = a_i$. Let $w^i = a_i, a_{i+1}, \ldots, a_n$, which is a suffix of w. We know from LTL semantics that $w \models \mathscr{X}\phi_i$ if $w^2 \models \phi_i$. Let us assume that $w^2 \models \phi_i$. Since, $\phi_i \equiv \psi_i$, we have $w^2 \models \psi_i$. Furthermore, $w(1) = a_1$ represents the traffic along an edge in the network and satisfies exactly one of $E_1, E_2, \ldots, E_m$, say E_1. Now, as we have described in Definition 4, conditionally composed constraints are evaluated as met or true, if the conditioning (i.e., the second) constraint is not met (which is the case for $E_2, \ldots, E_m$). Therefore,

$$w \models \psi_i|E_j, 2 \leq j \leq m \tag{4.5}$$

Now, since we know that E_1 is met, $\psi_i|E_1$ reduces to ψ_i being applied to traffic that has already satisfied E_1, and we already know that $w^2 \models \psi_i$, implying that

$$w \models \psi_i|E_1 \tag{4.6}$$

From equations 4.5 and 4.6, we have $w \models (\psi_i|E_1) \wedge \ldots \wedge (\psi_i|E_m)$. We can provide a similar argument in the other direction, proving the equivalence. □

Proposition 5. *(Until) Let ϕ_i and ϕ_j are LTL formulas, and ψ_i and ψ_j are their equivalent constraints. Let $\pi_1, \ldots, \pi_n$ be constraints representing each non-empty path that is a prefix of a path represented by ψ_i. Then*

$$\phi_i \mathscr{U}\, \phi_j \equiv (\pi_1|\neg\psi_j) \wedge \ldots \wedge (\pi_n|\neg\psi_j) \wedge (\psi_j|\neg\psi_i)$$

Proof. Let $w = a_1, a_2, \ldots, a_n$, where $a_i \in 2^{AP}$, be an ω-word or equivalently a traffic flow through the given network. Let $w(i) = a_i$. Let $w^i = a_i, a_{i+1}, \ldots, a_n$, which is a suffix of w. We know from LTL semantics that $w \models \phi_i \mathscr{U} \phi_j$ if $\exists k > 0$ such that $w^k \models \phi_j$ and for $0 < l < k, w^l \models \phi_i$. Intuitively, it means that ϕ_i must remain true until ϕ_j becomes true. We make use of the fact that we are restricted to finite length paths to enumerate each prefix sub-path in ψ_i. □

From Propositions 2, 3, 4, and 5, it inductively follows that any LTL formulas (formed through the use of the fundamental operators shown in Proposition 1) has an equivalent specification in our policy specification language. Therefore, we can claim that our policy specification language is at least as expressive as LTL.

CHAPTER 5

EXHAUSTIVE ANALYSIS

The algorithms described in this chapter produce an exhaustive list of all the inconsistencies and policy violations in the implementation of access control policy. For a given system, characterized by its network topology and firewall rule-sets, and a global policy specification, the products of exhaustive analysis are:

- A complete list of single firewall inconsistencies and redundancies for each firewall in the system.
- A complete list of inter-firewall inconsistencies among the firewalls in the system.
- A complete list of all the paths through the network that violate one or more global policy constraints (further identifying which constraints are violated by each such path).
- A list of root causes for the policy violations.

5.1 Identifying Inconsistencies

We start with algorithms to identify the intra-firewall and inter-firewall inconsistencies.

First, we consider the checks localized to ACLs on a single firewall. Since a firewall can rely on the filtering action of other firewalls to achieve policy conformance, local checks focus on checking inconsistency and inefficiency. The local check is performed after each rule is parsed, and just before the state is updated as defined in Equation 4.2.

The input to a localized rule-sub-graph is the entire set ($I = \Omega$), and $A_1 = D_1 = F_1 = \emptyset$. We process each rule in sequence based on its type:

For $< P, accept >$ rules:

1. $P_j \subseteq R_j \Rightarrow$ valid: This is a valid rule, as it defines an action for a new set of packets and does not overlap with any preceding rules.

2. $P_j \cap R_j = \emptyset \Rightarrow$ masked rule: This is an "error," as the rule will not match any packets and the accompanying action would never be taken. It has the following sub-cases:

   (a) $P_j \subseteq D_j \Rightarrow$ shadowing: The rule intended to accept some packets denied by preceding rules, thereby revealed to be a misconfiguration.

   (b) $P_j \cap D_j = \emptyset \Rightarrow$ redundancy: All packets have already been accepted by preceding rules or will not take this path.

   (c) Otherwise $\Rightarrow$ redundancy and correlation.

3. $P_j \nsubseteq R_j$ and $P_j \cap R_j \neq \emptyset \Rightarrow$ partially masked rule:

   (a) $P_j \cap D_j \neq \emptyset \Rightarrow$ correlation: Some of the packets this rule intended to accept have been denied by previous rules. This is a "warning" of potential misconfiguration.

   (b) $\forall x < j, \exists < P_x, deny >$ such that $P_x \subseteq P_j \Rightarrow$ generalization: Rule j is a generalization of rule x, since the latter matches a subset of the former's predicate but define a different action. This is a "warning" of potential misconfiguration.

   (c) $P_j \cap A_j \neq \emptyset$ and $\forall x < j, \exists < P_x, accept >$ such that $P_x \subseteq P_j \Rightarrow$ redundancy: If rule $< P_x, accept >$ is removed, all packets matching that rule can still be accepted by $< P_j, accept >$. This is an "error."

Similarly, for $< P, deny >$ rules:

1. $P_j \subseteq R_j \Rightarrow$ valid.

2. $P_j \cap R_j = \emptyset \Rightarrow$ masked rule, with following sub-cases:

   (a) $P_j \subseteq A_j \Rightarrow$ shadowing: This rule intended to deny some packets that have been accepted by preceding rules. There is a serious "error" in one of the rules.

   (b) $P_j \cap A_j = \emptyset \Rightarrow$ redundancy: All the packets have been denied by preceding rules or will not travel this path.

(c) Otherwise $\Rightarrow$ redundancy and correlation.

3. $P_j \nsubseteq R_j$ and $P_j \cap R_j \neq \emptyset \Rightarrow$ partially masked rule:

   (a) $P_j \cap A_j \neq \emptyset \Rightarrow$ correlation: Some of the packets that this rule intended to deny have been accepted by previous rules.

   (b) $\forall x < j, \exists < P_x, accept >$ such that $P_x \subseteq P_j \Rightarrow$ generalization: Rule j is a generalization of rule x, since the latter matches a subset of the former's predicate but define a different action. This is a "warning" of potential misconfiguration.

   (c) $P_j \cap D_j \neq \emptyset$ and $\forall x < j, \exists < P_x, deny >$ such that $P_x \subseteq P_j \Rightarrow$ redundancy: If rule $< P_x, deny >$ is removed, all packets matching that rule can still be denied by $< P_j, accept >$. This is an "error."

After passing the local checks, we perform distributed checks for networks of access control devices. We start from the top of the network-layer rule-graph and go downwards level-by-level. At the top, the input is the entire (universal) set ($I = \Omega$), and $A_1 = D_1 = F_1 = \emptyset$.

The inter-firewall consistency checks are performed much like the single-firewall rule-sub-graph checks, as follows.

For $< P, accept >$ rules:

1. $P \subseteq I \Rightarrow$ valid: This is not a redundancy, as it would be fore local checks.

2. $P \subseteq \neg I \Rightarrow$ shadowing: The rule shadowed by upstream ACLs as it is trying to accept packets that were denied on all reachable paths to it.

For $< P, deny >$ rules:

1. $P \subseteq I \Rightarrow$ increased security level.

2. $P \subseteq \neg I \Rightarrow$ potential redundancy: This represents either defense in depth or oversight; a "warning" is flagged for review.

5.2 Identifying Policy Violations

We start with a formal description of the base algorithm that interfaces rule-graph exploration with the policy specification. We then introduce various optimizations to the base algorithm, such as the incorporation of sub-path caching and data structures for attribute sets. We describe various heuristics for post-analysis for root cause identification, such as sort-threshold and min-cut.

The main algorithm for identifying policy violations is as follows. We assume a conflict-free global policy specification.

The global policy specification is used to calculate five traffic attribute sets for each node in the network. Let TAS_U represent the universal set of network traffic. So, for a node i, we calculate:

- $TAS^i_{allow\_must}$: traffic that must reach the node i. This is calculated from the $allow_{exact}$ constraints.
- $TAS^i_{allow\_should}$: traffic that is allowed to reach, but does not need to reach, node i. This is calculated from the $allow_{min}$ constraints.
- $TAS^i_{deny\_explicit}$: traffic that is explicitly prevented from reaching node i. This is calculated from the $deny$ constraints.
- $TAS^i_{allow} = TAS^i_{allow\_must} \bigcup TAS^i_{allow\_should}$: traffic that is allowed to reach the node i.
- $TAS^i_{deny} = (TAS_U - TAS^i_{allow}) \bigcup TAS^i_{deny\_explicit}$: traffic that must not reach the node i.

We also associate a traffic attribute set, $TAS^i_{reached}$, with each node, representing the traffic that has been found to be arriving at the node during our analysis. Initially, $\forall i, TAS^i_{reached} = \emptyset$.

Once we have calculated the distributed version of the global policy specification as described above, we perform an exploration of the rule-graph using a DFS-like algorithm. We perform such explorations several times, once with each node in the network as the initial source of the traffic. The exploration follows the algorithm described in Section 4.2.2. As noted in that section, the entire rule-graph is never explicitly generated; we generate only the portions related to the current path being explored. We use the same terminology and symbols used in Section 4.2.2, namely

- I: TAS representing the possible traffic arriving at an ACL.

- A_j: TAS representing the traffic accepted by rules 1 through $(j-1)$.
- D_j: TAS representing the traffic denied by rules 1 through $(j-1)$.
- R_j: TAS representing the traffic that can possibly be accepted by the j^{th} rule.

When arriving at a node representing a firewall, we first process the AAA-related ACLs and use the results from their exploration to divide the input TAS I into *authenticated* and *unauthenticated* input TAS's. When we arrive at an ACL, we travel through its constituent rules sequentially, computing the traffic attribute sets A_j, D_j, and R_j for each rule. After computing each A_j, we use that as the input TAS for downstream nodes in the rule-graph, i.e., the other ACLs and hosts that are reachable in a single hop through the network. If needed, we perform network address translation (NAT) based transformation before passing A_j as input to the downstream nodes, replacing the source or destination IP addresses appropriately. We also use information about the VPN overlays and routing tables when determining potential downstream nodes.

The algorithm recursively returns if the current R_j becomes empty, or a host is reached. If a host has been reached, we perform the following steps before returning.

- We check I, the traffic incident on the host, for compliance with TAS^i_{allow} and TAS^i_{deny} sets for the host. If either $(I - TAS^i_{allow})$ or $(I \cap TAS^i_{deny})$ is not empty, we update the global list of violations found with the current path, along with the violating traffic as well as the underlying constraints that produced the portions of TAS^i_{allow} and TAS^i_{deny} that were violated.
- We update the $TAS^i_{reached}$ set associated with the host to include the incident traffic. After the update, we check whether $TAS^i_{allow\_must} \subseteq TAS^i_{reached}$; if true, we set a Boolean indicator associated with the host, $allow\_must\_valid_i$, to be true, indicating that the $allow_{exact}$ constraints related to the host have been satisfied.

After we have completed the recursive DFS-like exploration with a given start node, we add all the nodes for which the $allow\_must\_valid_i$ indicator is still false to the global list of violations. Once we have performed the exploration with each potential starting node, we return the global list of violations.

We make some additional optimizations to the algorithm described above.

- *Using sub-network TAS as initial source*: Instead of performing multiple runs of the exploration algorithm with each host serving as the starting point in turn, we use the TAS of each LAN-based sub-network as the initial source. Each time a violation is found, we disambiguate the initial source IP address based on the particular constraints that were violated.

- *Using cached computations for memoization*: We cache the various set-theoretic computation chains at each ACL in a lookup table indexed by the attributes of the incident TAS, I, and the A_j's for each rule in the ACL. Then each time we progress to a rule within the ACL while exploring the rule-graph, we use memoization to avoid repeated computation, by performing efficient lookup operations to find identical or near-identical (at most one dimension of the relevant interval tree differs) past calculations. Those cached results can then be used directly (in case of an identical match) or with minor tweaks (in case of a near-identical match). This memoization-based optimization significantly increases the memory requirements for our algorithm, but results in a big performance boost, particularly for networks with a large number of firewalls and sub-networks. The additional memory usage is not a major concern since the rule-graph is never explicitly modeled in its entirety at any stage of the algorithm. If the memory usage does become a concern, we provide the option of only caching the results of the last n computation chains at each ACL, where n can be reduced to lower the memory usage.

The final output of the analysis is a list of paths that violate one or more global access policy constraints. Each identified *violating path* further indicates the subset of constraints it violated and the attributes of the ending traffic that resulted in the violation. The paths typically begin at a host and end at a host, and include in the middle one or more rules matched in intervening firewalls (with only a single rule from any given firewall).

Even a few misconfigurations in firewall rule-sets can result in several violating paths. While they can provide a comprehensive view of the problems in the network, it would be useful to precisely identify the misconfigured rules that caused the violations. We provide two mechanisms for post-analysis root case identification.

Sort-threshold: We sort all the individual firewall rules found in all the violating paths by the

frequency of their occurrence. We report the rules with the frequency of occurrence above a user-specified threshold as the most likely root causes.

Min-cut: For each connected component in the sub-graph formed by the union of all the violating paths through the rule-graph, we identify the minimal set of internal vertices (the internal vertices represent various firewall rules that are part of one or more violating paths) that can be removed to make each connected component disconnected. We model this problem as a network flow problem with multiple sources and sinks, and the capacity and flow across each edge both being 1. We use a version of the minimum cut algorithm described by Hao and Orlin in [45], as it was shown to be the most robust performer in [46].

Sort-threshold is very fast, but is not guaranteed to be accurate. On the other hand, *min-cut* can be considerably more expensive computationally (depending on the size of the sub-graph formed by the union of violating paths), but is guaranteed to be accurate. Our implementation of the exhaustive analysis algorithms in the NetAPT tool provides users the option to select either method for root cause identification. In practice, as noted in Chapter 9, during the application of NetAPT to various test cases and industry field tests on networks of varying size and complexity, we have found that *sort-threshold* with a threshold parameter of 10 is sufficient to identify all the root causes.

5.3 Analytic Evaluation

We now provide an analysis of the space and time complexity of the algorithm (as a function of the topology, the policy and the number of misconfigurations). We analyze the effect of optimizations on the base algorithm on (worst case) space and time requirements. We provide analytic comparison of our algorithms with vanilla model-checking.

The order of growth for a graph algorithm is usually expressed as a function of $|V|$, the number of vertices, and $|E|$, the number of edges.

The order of growth for BFS is $O(|V|+|E|)$, which is a convenient way to say that the run time grows in proportion to either $|V|$ or $|E|$, whichever is "bigger."

To see why, think about these four operations:

Adding a vertex to the queue: this happens once for each vertex, so the total cost is in $O(|V|)$.

Removing a vertex from the queue: this happens once for each vertex, so the total cost is in $O(|V|)$.

Marking a vertex "visited": this happens once for each vertex, so the total cost is in $O(|V|)$.

Checking whether a vertex is marked: this happens once each edge, so the total cost is in $O(|E|)$.

Adding them up, we get $O(|V| + |E|)$. If we know the relationship between $|V|$ and $|E|$, we can simplify this expression. For example, in a regular graph, the number of edges is in $O(|V|)$ so BFS is linear in $|V|$. In a complete graph, the number of edges is in $O(|V|^2)$ so BFS is quadratic in $|V|$.

Of course, this analysis is based on the assumption that all four operations—adding and removing vertices, marking and checking marks—are constant time.

Marking vertices is easy. You can add an attribute to the `Vertex` objects or put the marked ones in a set and use the `in` operator.

But making a first-in-first-out (FIFO) queue that can add and remove vertices in constant time turns out to be non-trivial.

We start with a simplified version of the algorithm that considers all edges the same length. The more general version works with any non-negative edge lengths.

The simplified version is similar to the breadth-first search except that instead of marking visited nodes, we label them with their distance from the source. Initially all nodes are labeled with an infinite distance. Like a breadth-first search, Dijkstra's algorithm uses a queue of discovered unvisited nodes.

1. Give the source node distance 0 and add it to the queue. Give the other nodes infinite distance.

2. Remove a vertex from the queue and assign its distance to d. Find the vertices it is connected to. For each connected vertex with infinite distance, replace the distance with $d + 1$ and add it to the queue.

3. If the queue is not empty, go back to Step 2.

The first time we execute Step 2, the only node in the queue has distance 0. The second time, the queue contains all nodes with distance 1. Once those nodes are processed, the queue contains all nodes with distance 2, and so on.

So when a node is discovered for the first time, it is labeled with the distance $d+1$, which is the shortest path to that node. It is not possible that we will discover a shorter path later, because if there were a shorter path, we would have discovered it sooner. That is not a proof of the correctness of the algorithm, but it sketches the structure of the proof by contradiction.

In the more general case, where the edges have different lengths, it is possible to discover a shorter path after we have discovered a longer path, so a little more work is needed.

Most arithmetic operations are constant time; multiplication usually takes longer than addition and subtraction, and division takes even longer, but these run times don't depend on the magnitude of the operands. Very large integers are an exception; in that case the run time increases linearly with the number of digits.

Indexing operations—reading or writing elements in a sequence or dictionary—are also constant time, regardless of the size of the data structure.

The string method `join` is usually faster because it is linear in the total length of the strings.

As a rule of thumb, if the body of a loop is in $O(n^a)$ then the whole loop is in $O(n^{a+1})$. The exception is if we can show that the loop exits after a constant number of iterations. If a loop runs k times regardless of n, then the loop is in $O(n^a)$, even for large k. Multiplying by k doesn't change the order of growth, but neither does dividing. So if the body of a loop is in $O(n^a)$ and it runs n/k times, the loop is in $O(n^{a+1})$, even for large k. Most string and tuple operations are linear, except indexing and `len`, which are constant time. The built-in functions `min` and `max` are linear. The run-time of a slice operation is proportional to the length of the output, but independent of the size of the input.

All string methods are linear, but if the lengths of the strings are bounded by a constant—for example, operations on single characters—they are considered constant time. The `in` operator for sequences uses a linear search; so do string methods like `find` and `count`.

If the elements of the sequence are in order, we use a *bisection search*, which is $O(\log n)$. Bisection search is similar to the algorithm we probably use to look a word up in a dictionary (a real dictionary, not the data structure). Instead of starting at the beginning and checking each

item in order, we start with the item in the middle and check whether the word we are looking for comes before or after. If it comes before, then we search the first half of the sequence. Otherwise we search the second half. Either way, we cut the number of remaining items in half.

`add` appends a key-value tuple to the list of items, which takes constant time.

`get` uses a `for` loop to search the list: if it finds the target key it returns the corresponding value; otherwise it raises a `KeyError`. So `get` is linear.

An alternative is to keep the list sorted by key. Then `get` could use a bisection search, which is $O(\log n)$. But inserting a new item in the middle of a list is linear, so this might not be the best option. There are other data structures, namely Red-Black Trees, that can implement `add` and `get` in log time, but that's still not as good as a hashtable, so let's move on.

One way to improve `LinearMap` is to break the list of key-value pairs into smaller lists. Here's an implementation called `BetterMap`, which is a list of 100 LinearMaps. As we'll see in a second, the order of growth for `get` is still linear, but `BetterMap` is a step on the path toward hashtables:

`find_map` uses the modulus operator to wrap the hash values into the range from 0 to `len (self.maps)`, so the result is a legal index into the list. Of course, this means that many different hash values will wrap onto the same index. But if the hash function spreads things out pretty evenly (which is what hash functions are designed to do), then we expect $n/100$ items per LinearMap.

Since the run time of `LinearMap.get` is proportional to the number of items, we expect BetterMap to be about 100 times faster than LinearMap. The order of growth is still linear, but the leading coefficient is smaller. That's nice, but still not as good as a hashtable.

Here (finally) is the crucial idea that makes hashtables fast: if we can keep the maximum length of the LinearMaps bounded, `LinearMap.get` is constant time. All we have to do is keep track of the number of items and when the number of items per LinearMap exceeds a threshold, resize the hashtable by adding more LinearMaps. `get` just dispatches to `BetterMap`. The real work happens in `add`, which checks the number of items and the size of the `BetterMap`: if they are equal, the average number of items per LinearMap is 1, so it calls `resize`.

`resize` make a new `BetterMap`, twice as big as the previous one, and then "rehashes" the items from the old map to the new.

Rehashing is necessary because changing the number of LinearMaps changes the denominator of the modulus operator in `find_map`. That means that some objects that used to wrap into the

same LinearMap will get split up (which is what we wanted, right?).

Rehashing is linear, so `resize` is linear, which might seem bad, since we indicated that `add` would be constant time. But remember that we don't have to resize every time, so `add` is usually constant time and only occasionally linear. The total amount of work to run `add` n times is proportional to n, so the average time of each `add` is constant time.

CHAPTER 6

STATISTICAL ANALYSIS

When the networked system (considered together with the relevant components of the IT network) being analyzed is large and deep, the combinatorics of the possible interactions among the access control devices and mechanisms may render a comprehensive exhaustive analysis computationally impossible. The underlying rule graph would simply contain too many paths to allow an exhaustive exploration, especially when the actual paths of interest (i.e., the violations) form a very small subset of the set of all possible paths through the rule graph.

To handle this challenge, we incorporate a statistical analysis approach that produces a sample (likely incomplete) set of policy violations, and a quantitative estimate of the remainder. The latter may take the form of estimates of the total number of violations, the fraction of all traffic that violates global policy, or the probability that there are no violations given that none were discovered during a specified amount of time after the analysis was performed.

We obtain statistically valid estimates of such quantitative measures without actually exploring the entire rule graph. We do so through repeated and random exploration of an extended version of the rule graph to sample a few paths, using mathematically formulated heuristics to guide the choice at each step of the exploration towards the likely sources of policy violation. The desired set of metrics is calculated for each sampled path, and then we use "importance sampling" [47] to remove the bias introduced by the guidance heuristic and obtain unbiased estimators of the metrics. As the analysis continues, the user can watch the progress of the analysis, including the convergence of the chosen set of metrics, using the graphical front-end. The process can be continued until a user-specified relative error bound is reached or a user-specified fraction of the rule graph has been explored. The user can also abort the analysis at any time.

The main technical objective of this aspect of our work is to develop metrics that measure security compliance, use importance sampling to estimate them, and prove critical properties of the im-

portance sampling heuristics. Importance sampling biases sampling towards paths that contribute the most to the metric. While this technique cannot prove that no violation exists, or guarantee that it will find every violation, it can provide a very good overall assessment of compliance in systems that otherwise could not be analyzed.

6.1 Estimation of Security Compliance Metric

We first introduce our ideas about estimating compliance in the context of firewall (that is, firewall-only) systems. We define a metric that captures an implementation's compliance with global policy, but do so in such a way that a user can emphasize the hosts, traffic, and policy rules that matter most. We then show how to use importance sampling to estimate that metric, report on preliminary results we've achieved, and outline our research plan in the area of estimation.

6.1.1 Metric Definition

It is commonly recognized that formulation of good security metrics is a key problem. One area in which our work has technical merit is in identification of a good metric for our specific application domain.

Let $\Pi = (\Sigma, \Psi)$ be the global policy specification, where Ψ is the set of global policy rules (constraints). Consider any given traffic source V_s, and the full set of attributes of a potential packet p it would send. Different packets could have different levels of importance; for example, some might be used in critical network services. Notionally, we attach some level of importance $w(V_s, p)$ to this packet, understanding that ultimately this weight is subjective; while natural default values are easy to introduce, any specialization has to be provided by a user. Likewise, the violation of different policy rules might have differing levels of importance. For example, in a domain whose operations are federally regulated, some violations may have associated fines, while others will not. Correspondingly, if p violates one or more rules in Ψ, we define the violation score $v(\Psi, V_s, p) \geq 0$ to numerically assess the strength of the violation; again, while $v(\Psi, V_s, p) = 1$ is an intuitive default value, a user can tailor the function to encode special needs.

Combining the above definitions, we come up with a packet's compliance score, $c(\Psi, V_s, p) =$

$w(V_s, p) * v(\Psi, V_s, p)$. We use different sums of those scores. Considering all packets that source V_s might produce, we are led to V_s's *security compliance metric* ($\mathscr{G}$).

$$\mathscr{G}(\Psi, V_s) = \sum_{p \text{ that } V_s \text{ can produce}} c(\Psi, V_s, p).$$

At other points, we will refer to the compliance of all packets that follow a common path $\mathscr{P}$ through the rule-graph. If the common source of those packets is defined by $\mathscr{P}_s$, then we obtain path $\mathscr{P}$'s security compliance metric:

$$\mathscr{G}(\Psi, \mathscr{P}) = \sum_{p \text{ following } \mathscr{P}} c(R, \mathscr{P}_s, p). \tag{6.1}$$

We are ultimately interested in a system's overall security compliance score. We can weight different sources to reflect relative importance; the weight given to source V_s is denoted by q_s. The system security compliance score is then

$$\mathscr{G}(\Psi) = \sum_{\text{sources } V_s} q_s \mathscr{G}(\Psi, V_s). \tag{6.2}$$

Given the various points where weights are applied, $\mathscr{G}(\Psi)$ is a metric whose definition encompasses several natural interpretations, depending on the weights chosen. They include the following:

- *A packet's mean violation score*, where the probability distribution may be uniform over the space of all possible packets or be based on a frequency distribution created from historical data.
- *The fraction of traffic that violates global policy*; this is either the fraction associated with a given source s, or the fraction taken over the entire system. The fractions can be with respect to IP space, or with respect to frequencies based on historical data.
- *Both interpretations above, but limited to violation of a subset of global policy rules.*
- *The number of paths through the rule-graph that lead to violation of any rule in some subset*

of all rules (including, e.g., the set of all rules).

6.1.2 Review of Importance Sampling

To give a metric meaning, we have to quantify it. A key element of this dissertation's intellectual merit is our demonstration that the process of randomly selecting paths through the rule-graph, when intelligently guided by importance sampling, can efficiently estimate $\mathscr{G}(\Psi)$. The basic idea is to avoid wasting computational effort on paths whose compliance scores are low relative to others. Biasing the random selection towards paths on which the metric has the largest value maximizes the "return on investment" of computational work dedicated to that sample.

That general method has a long history in contexts as diverse as estimation of integrals in high dimensions, chemistry, communications, reliability, and economics. See [48, 49, 50] for a representative selection. We have used it ourselves to estimate measures of an attacker's success against a system with known vulnerabilities [33]. The challenge is that the sampling strategies are necessarily domain-specific. The strategies one uses to estimate the value of rate constants in chemical reactions depend on the chemical equations being studied; the strategies one uses to estimate the probability of buffer overflow in a network switch are completely different. Each strategy must take advantage of what is known about the domain, in order to bias sampling toward system states that have the largest impact on the metric being estimated.

A simple example illustrates importance sampling. Let X be a random variable with probability density function u, let $A \subset \mathbb{R}$, and let $\gamma = \Pr\{X \in A\}$. Define the *indicator random variable* $1_{\{X \in A\}}$ as follows: sample x. If $x \in A$ set $1_{\{X \in A\}} = 1$, otherwise set $1_{\{X \in A\}} = 0$. It follows that $\gamma = E_u[1_{\{X \in A\}}]$, with the expectation taken with respect to u.

A naive Monte Carlo estimate of γ generates samples $X_1, X_2, \ldots, X_N$ using the density u, and constructs estimator

$$\hat{\gamma}_N = \frac{1}{N} \sum_{i=1}^{N} 1_{\{X_i \in A\}}.$$

$\hat{\gamma}_N$ is *unbiased*, in the sense that $E_u[\hat{\gamma}_N] = \gamma$.

The width of the associated confidence interval is proportional to the standard deviation, which is $\sqrt{\gamma(1-\gamma)/N}$. For $\hat{\gamma}_N$ to have numerical significance relative to its uncertainty, we need for

the *relative error* $\sqrt{\gamma(1-\gamma)/N}/\gamma$ to be small. Problems arise when γ is small; a very large N is needed to continue to maintain a small relative error.

Importance sampling achieves small relative error using significantly smaller N than naive sampling uses. We define another probability density function $u'(x)$, with $u'(x) > 0$ for all $x \in A$ such that $u(x) > 0$. Then, $\gamma = E_{u'}[1_{\{X \in A\}} L_{p'}(X)]$, where $L_{p'}(x) = u(x)/u'(x)$ is known as the likelihood ratio, and the expectation is taken with respect to u'. We generate samples $X_1', X_2', \ldots, X_N'$, using u'. Then

$$\hat{\gamma}_N = \frac{1}{N} \sum_{i=1}^{N} 1_{\{X_i \in A\}} L_{p'}(X_i)$$

is an unbiased estimator of γ. It has been shown [47] that to reduce the variance of the estimator $\hat{\gamma}_N$, we need to make the likelihood ratio $u(x)/u'(x)$ small on the set A, e.g., u' should bias towards $x \in A$.

The aim, then, is that through the use of importance sampling, the relative error associated with N samples will be smaller than the relative error for N samples from a normal Monte Carlo scheme. That implies greater accuracy for the estimator, meaning significantly better results for the same number of samples.

An identical discussion applies for estimating $\alpha = E_u[W(X)1_{\{X \in A\}}]$, where W is some function of the random variable.

6.1.3 Application to Security Compliance Metric $\mathscr{G}(\Psi)$

Importance sampling is a general technique, but the challenge in using it lies in applying the general theory to a specific context. We must prove that the random variable whose mean we can estimate by Monte Carlo sampling is identical to $\mathscr{G}(\Psi)$; we must prove that the strategy we use to bias sampling for importance sampling is correct, in the sense of estimating $\mathscr{G}(\Psi)$ as well; we also need to prove that the biased sampling technique is computationally effective.

Each Monte Carlo trial in our approach is a random sample of a path through the rule-graph that ends either in delivery to a host, or in rejection at a firewall; each trial thus has a path compliance score (recall Equation 6.1). The path chosen can be represented by the random variable X in the example above, and the event of the path violating global policy can be represented by an indicator

$1_{\{X \in A\}}$. Probability density function u reflects the random selection of rules that define the path. The path compliance value is can be represented by the function $W(x)$.

Sample Space: In order to create unbiased estimators, we have to put random path selection on a firm mathematical footing. The first step is to identify a useful sample space, or set of events that can occur during sampling.

Let $\mathscr{S}$ be the set of all rule-graph nodes. Let $\mathscr{S}_0$ denote the set of root nodes, i.e., the traffic sources. A *sample sequence* is any infinite sequence $\{d_i \in \mathscr{S}, i > 0\}$. There is no supposition of any kind of structure, except that each element of the sequence is drawn from $\mathscr{S}$. We use the term *valid prefixes* to describe legitimate complete paths in the rule graph, starting in $\mathscr{S}_0$ and terminating in a final policy decision. "Legitimate" here means that there exists a packet with attributes such that the firewall rules force it to traverse that path.

We say that a valid prefix is *interesting* if the corresponding path conflicts with global policy, and let $\mathscr{V}$ be the set of all interesting prefixes. Let the sample space Ω be the set of all sample sequences, including infinite ones. We can enumerate the nodes in $\mathscr{S}$, and transform each index into a single digit in the base $|\mathscr{S}|$ number system. That means that we can establish a one-to-one correspondence between every real number in $[0,1]$ and every element of Ω; given sample sequence $d_1, d_2, \ldots$ (where the d_i are the base $|\mathscr{S}|$ digits representing the nodes), we map to the base $|\mathscr{S}|$ the real number $(0.d_1 d_2 d_3 \ldots)_{|\mathscr{S}|}$, and vice versa. We can thus associate a uniformly random sample U from $[0,1]$ with a unique element of Ω. Equivalently, we define a probability measure on Ω such that all sample sequences are equally likely. Henceforth, we drop the subscript $|\mathscr{S}|$, and all real numbers are in base $|\mathscr{S}|$.

The set of all sample sequences with the interesting prefix $(0.d_1 d_2 \ldots d_l)$ (i.e., all sample sequences whose first l elements are identical to $d_1, d_2, \ldots, d_l$) form a closed interval of length $|\mathscr{S}|^{-l}$ starting at $(0.d_1 d_2 \ldots d_l)$. For every interesting prefix v, the corresponding interval is denoted by I_v, and its length by l_v. Notice that if $v, w \in \mathscr{V}$, then $I_v \cap I_w = \emptyset$. It can be seen that $\mathscr{A} = \bigcup_{v \in \mathscr{V}} I_v$ forms a finite union of disjoint intervals. Hence, the probability of a uniform sample $u \in [0,1]$ also having $u \in \mathscr{A}$ is the sum of the lengths of the constituent intervals. Due to the large size of Ω, that total probability would be very small, and membership in $\mathscr{V}$ would be a rare event under uniform sampling on $[0,1]$.

Unbiased Estimation: Our development of Ω sets the stage for constructing an unbiased estimator of $\mathscr{G}(R)$. Given any value $x \in [0,1]$, we can identify the prefix v_x that is in one-to-one correspondence with x and hence identify the path $\mathscr{P}_{v_x}$ represented by that prefix, and determine (i) the length l_{v_x} of that prefix, and (ii) whether the prefix is interesting. For interesting prefixes v_x, we compute (recalling Equation 6.1) $M(x) = |\mathscr{S}|^{l_{v_x}} \mathscr{G}(R, \mathscr{P}_{v_x})$. Therefore, we have

$$\begin{aligned} E_u[M(X)] &\equiv E_u[M(X)1_{\{X\in\mathscr{A}\}}] = \int_{-\infty}^{\infty} M(X)1_{\{x\in\mathscr{A}\}}u(x)\,dx \\ &= \int_{x\in\mathscr{A}} M(x)u(x)\,dx \\ &= \sum_{v\in\mathscr{V}} \left(\int_{x\in I_v} M(x)u(x)\,dx \right) \end{aligned}$$

$$\begin{aligned} &= \sum_{v\in\mathscr{V}} \left(\int_{x\in I_v} M(x)\,dx \right) = \sum_{v\in\mathscr{V}} \left(|\mathscr{S}|^{l_v} \mathscr{G}(R, \mathscr{P}_v) \int_{x\in I_v} dx \right) \\ &= \sum_{v\in\mathscr{V}} |\mathscr{S}|^{l_v} \mathscr{G}(R, \mathscr{P}_v) |\mathscr{S}|^{-l_v} = \mathscr{G}(R) \end{aligned}$$

where the introduction of summation is possible because $\mathscr{A}$ is a finite union of disjoint I_v's. Hence $E_u[M(X)]$ is an unbiased estimator of $\mathscr{G}(\Psi)$.

Change of Measure: If ordinary Monte Carlo sampling is used on Ω, the standard unbiased estimator would be $\frac{1}{N}\sum_{i=1}^{N} M(X_i)$, where $X_1, X_2, \ldots, X_N$ are N random samples drawn from $[0,1]$ (or, equivalently, from Ω) with density $u(x)$. However, estimating $E_u[M(X)]$ this way is extremely inefficient, as most samples would not correspond to valid prefixes, let alone interesting ones. The value of this formulation is that *given* an x that corresponds to a valid prefix, computation of the probability mass it represents under uniform sampling is straightforward. Each sample under the importance sampling approach randomly generates a path $d_1, d_2, d_3, \ldots$ through the rule graph, stopping when a final policy action is applied. The path chosen corresponds automatically to a valid prefix. The probability distribution used to select d_i given $d_1, d_2, \ldots, d_{i-1}$ is the biased sampling strategy. Whatever that strategy is, $q_{v,i}$ denotes the probability of selecting d_i, given the prior selections $d_1, d_2, \ldots, d_{i-1}$, with the understanding that $v = (d_1, d_2, \ldots, d_i)$. Mapping a selected sequence back into a real number x gives rise to a new probability density function u' on

$[0,1]$, driven by our biasing strategy. As we have shown in [33], the above change of measure is valid, and the value of $u'(x)$ for $x \in I_v$ for some interesting prefix v is

$$u'(x) = \frac{\prod_{i=1}^{l_v} q_{v,i}}{\text{length of } I_v} = \frac{\prod_{i=1}^{l_v} q_{v,i}}{|\mathscr{S}|^{-l_v}} = |\mathscr{S}|^{l_v} \prod_{i=1}^{l_v} q_{v,i} \quad \text{for all } x \in I_v. \tag{6.3}$$

Therefore, using importance sampling, we sample N valid prefixes according to our biasing strategy, map each into base $|\mathscr{S}|$ numbers $X_1, X_2, \ldots, X_N$, and construct the estimator

$$(1/N) \sum_{i=1}^{N} \left(\frac{u(X_i)}{u'(X_i)} \right) M(X_i).$$

Despite the large numbers symbolically expressed above (e.g., $|\mathscr{S}|^{l_v}$), the actual computations used in constructing this estimator can be done with much smaller values, avoiding the risk of round-off and precision errors.

6.2 Estimation with No Exceptions Identified

Estimation with no exceptions identified is similar to verification of properties defined in logics such as LTL if those properties are composed strictly of the *always* temporal modal operators ($\mathscr{G}$). For such properties, verification can be decomposed and any property of the form $\mathscr{G}\phi$ can be verified for any sub-model. For example, the approach can be used in statistical analysis of rule-graphs for verification of network access control policy implementations. In particular, the property ϕ can be the global access policy for the networked system being analyzed. During the statistical analysis of the rule-graph for the system, we use some biasing heuristics (and importance sampling) to direct our exploration of the rule-graph in certain ways. The biasing heuristics typically direct the exploration towards the "important" nodes in the rule-graph (nodes explicitly mentioned as being relevant in the global policy, or nodes representing recent configuration changes). On reaching the stopping criteria for the exploration, if we found some violations or counterexamples, we can use importance sampling to provide unbiased estimates of the total number of exceptions in the entire rule-graph.

However, that does not work if no exceptions were found during the exploration. In such a

situation, we can use the above approach to provide (under certain assumptions), with certain confidence, the likelihood that there are indeed no exceptions in the system. As an example, we can use importance sampling to generate an estimate of the total number of paths in the rule-graph ($\hat{N}$). We already know the exact number of paths that we explored before reaching the stopping criteria (e). If m is the total number of unexceptional paths in the rule-graph, the probability that the number of unexceptional paths seen so far is k is given by the *hypergeometric distribution*:

$$P_k = \frac{\binom{m}{k}\binom{\hat{N}-m}{e-k}}{\binom{\hat{N}}{e}}$$

In this scenario, $k = e$, as all the paths we have explored were unexceptional. The probability that of the remaining unexplored paths, i are unexceptional is given by:

$$P(k = e|m = i) = \frac{\binom{i}{e}\binom{\hat{N}-i}{0}}{\binom{\hat{N}}{e}} = \frac{\binom{i}{e}}{\binom{\hat{N}}{e}}$$

Using that, we obtain:

$$P(k = e) = \sum_{i=e}^{i=\hat{N}} P(k = e|m = i)$$

Then, our desired probability would be (from the Bayes theorem):

$$P(m = \hat{N}|k = n) = \frac{P(k = n|m = \hat{N})P(m = \hat{N})}{P(k = n)}$$

The only unknown in the above equation is $P(m = \hat{N})$, which we can estimate by making certain assumptions about the system and restricting our analysis to the classes of systems for which those assumptions are true.

CHAPTER 7

TOPOLOGY INFERENCE

Before the access control implemented in a networked system can be analyzed by our exhaustive or statistical analysis algorithms, we need a detailed map of the networked system, indicating how various devices are connected. It must include the information identified in Section 4 and Figure 4.3. However, in practice, network administrators frequently do not have this information with the necessary precision and detail. As a result, we have developed and implemented a set of algorithms that infer the underlying network topology from a variety of indirect information sources, such as the native configuration files for various Layer 3 devices, and output the topology in XML that conforms to the topology schema indicated in Figure 4.3.

7.1 Overall Topology Inference Framework

The main elements of the overall framework are:

1. *Topology database*: This database stores information about the myriad topology elements collected from different input sources. The input sources can potentially include firewall configuration files, dumps from nmap scans, and log files from routers and switches. The framework includes scripts that provide the necessary interface to the database, while managing its internal integrity and conformance to the relational DB schema set out for it.

2. *Input scripts*: These scripts provide a modular and extensible way to parse the raw information from the various input sources and populate the topology database.

3. *Output scripts*: These scripts read the information contained in the topology database and produce "views" of the information in various formats that can be easily visualized and

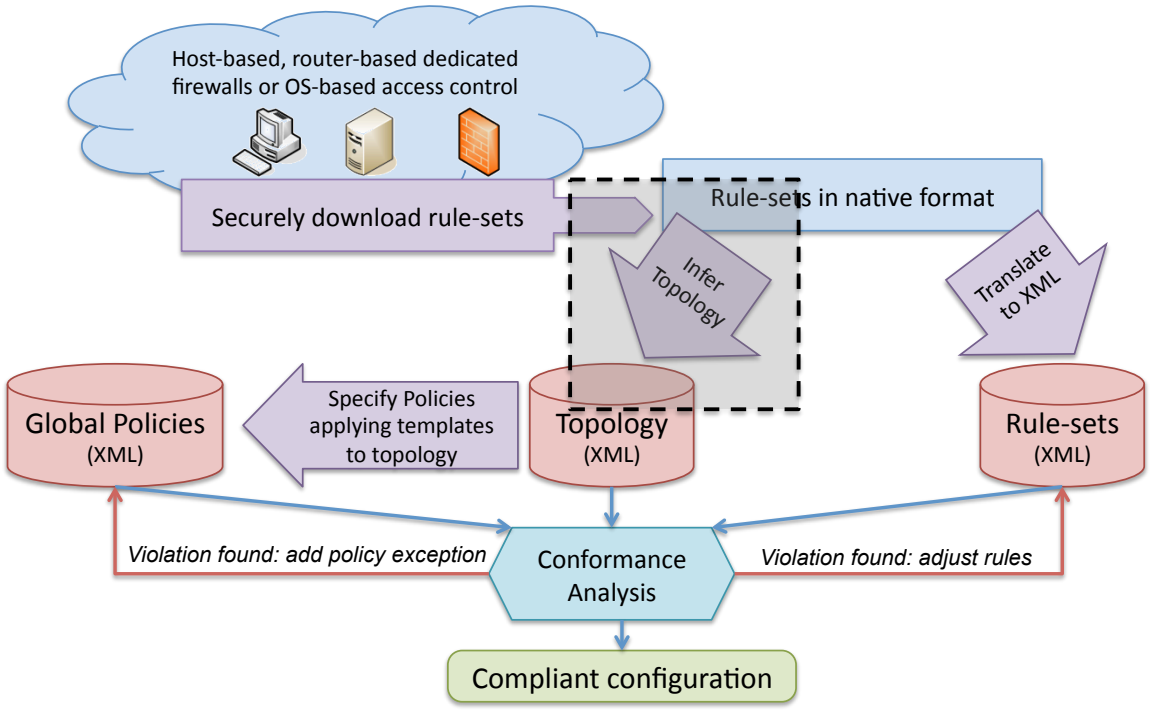

Figure 7.1: Topology Inference as Part of Overall Framework

processed. These include formats like GraphML, Graphviz, and our own topology XML schema.

Currently, our input scripts have primarily focused on firewall configurations as source of connectivity information. A firewall's configuration includes elements of connectivity information such as:

- Classless Inter-Domain Routing (CIDR) descriptions of the sub-networks that face the firewall's interfaces.
- "route" statements indicating explicit information about the traffic routed through the firewall.
- VPN descriptions characterizing the terminal points, security associations and other details about VPN tunnels.
- Access control list rules that indicate the traffic passing through the firewall.

- Object groups characterizing (and perhaps labeling) hosts, sub-networks, services, among others.

Our database-based approach uses the above connectivity information to "grow" the knowledge of the topology. We identify sub-networks that are connected to firewalls and "merge" similar sub-networks to build additional connections.

Firewall configurations contain elements of extant host IP addresses, such as "name" statements; ACL, object group and other statements that directly refer to a host IP address; and reverse DNS lookups. We populate the discovered sub-networks with the discovered host IDs.

Our framework is based on Sandia National Lab's ANTFARM tool [51]. We have made extensive additions and modifications to the base functionality in ANTFARM.

We add support for layer 3 switches and routers, provide sophisticated network merge algorithms that include support for complex merge disambiguation. The network merge disambiguation tries to answer the following question: if the address of the network N_1 is a subset of the address of network N_2, should N_1 be considered a part of N_2 and be merged into N_2? Our merge algorithm allows for:

- *Always merge*: we merge whenever possible.

- *Description-based merge*: we use meta-information stored in the interface blocks of the firewall configuration files to determine if the networks presented for merge are actually the same network.

The merging algorithm is tunable through user-specified input that determines how public and private networks should be merged (e.g., always merge networks with public IP addresses and use description-based merging for networks with private IP addresses).

We have developed custom input parsers that support a wide variety of features found in the most popular models of firewalls. Examples include:

- Information in the rules (access-list and access-group commands).

- Support for static and dynamic NATs, and virtual hosts.

- Support for Layer 3 switches and routers.

- Support for IPSec Security Associations and VPN tunnels (expands existing ANTFARM code).
- Support for nested object group definitions, etc.
- Support for meta-information required for description-based merging

We use completely custom output scripts that generate the inferred topology encoded with our topology XML schema (Figure 4.3).

A much more detailed description and analysis of the algorithms used for topology inference is provided in Appendix A.

CHAPTER 8

FRAMEWORK IMPLEMENTATION: NETWORK ACCESS POLICY TOOL

We have implemented the framework and algorithms described so far in the form of the Network Access Policy Tool (NetAPT).

8.1 NetAPT Architecture

As shown in Figure 8.1, NetAPT has two independent components:

- a graphical front-end (or management console), written in Java Swing, that system administrators can run from their workstations and use to provide information about the basic network topology, to enter specifications of the global access policy (or subsets thereof), and to set up analyses; and
- an analysis engine, written in C++, which captures the system state and analyzes that information with respect to policy specifications.

As indicated in Figure 8.1, the two components of the tool communicate securely over the network using TCP/TLS protocols. This segregation of functionality makes it possible to use the analysis engine as an appliance that can be plugged into a suitable spot in the network, from which it can establish secure connections to the access control elements present on the network and capture the relevant configuration information. The actual interface between the tool and the user (the front-end) can then be freely placed where it is convenient for the user.

As shown in the figure, the user first provides a description of the network topology using the graphical front-end. That can be done with the easy-to-use drawing tools included with the front-end, or via text files that can be read by the front-end. The information to provide includes all the relevant access control elements (firewalls, proxy servers, wireless access points); details for each

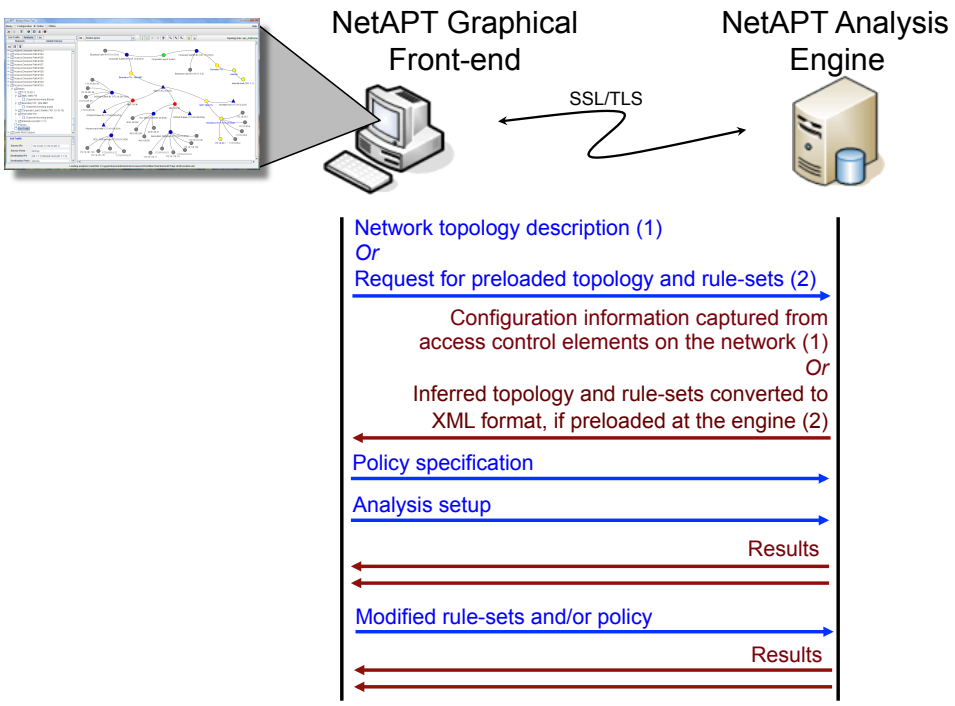

Figure 8.1: NetAPT Architecture

element, such as its IP address(es) and parameters that the analysis engine can use to establish secure connections to the element to capture its configuration; and all the network interactions between the listed elements. Entities can be grouped together to indicate sub-networks and LANs, and properties can be specified for such groupings. The front-end converts the visual topology representation provided by the user to an internal XML representation. We use XML as our underlying language for internal representation for most of the information in the tool, which provides the added benefit that we may interface with third-party tools and applications simply by providing wrappers that output the XML conforming to our schemas. It is also possible to browse and select from the list of topology descriptions previously cached remotely at the analysis engine. In either case, once the topology has been decided, the information is sent over the network to the analysis engine. The analysis engine uses that information to securely obtain the snapshot of the access control policy implementation, in the form of configuration files from the various devices indicated in the topology description (Figure 4.3). It also indexes and caches the topology description and captured configuration information (both of which are encrypted before storage for added security). The captured information from all the different sources is then converted to XML (using

a unified schema for firewall rules and another for OS-based mechanisms), and the XML is sent back to the front-end, where the user can then highlight various elements in the topology diagram and look at their current configurations (e.g., rule-sets for a firewall).

The user then specifies the global access constraints, which are a subset of the possible comprehensive access policy. They indicate some intended behavior against which the user wishes to check the policy implementation for compliance. The user uses a graphical front-end to specify constraints for the various elements or groupings of elements, and the tool considers their conjunction. The individual constraints can express positive as well as negative assertions about the nature of traffic, roles, user-classes, or applications that can access a particular set of resources (hosts, specific files or applications, and so forth). In other words, we can express things that should never happen as well as things that must be allowed. Once specified, the constraints are converted into an internal XML representation. This approach offers the freedom of being able to use third-party technologies, such as a variety of modal logics, to specify the global access constraints.

The next step for the user is to set up the analysis. He or she can select either an exhaustive analysis or a statistical analysis, following which the front-end sends the policy specification and setup information for the analysis to the analysis engine. The analysis engine sends back the results as it obtains them, giving the user the option of aborting the analysis at any stage and doing post-analysis on the partial results. The user can manipulate, filter, or navigate through the results, if any violations are found, to visualize the problem and diagnose the key misconfigurations behind the violations. A variety of post-analysis techniques are available in the front-end to help the user identify the likely root causes of the violations. Once the user has some possibilities in mind, hypothetical changes can be analyzed using the front-end; the user can make proposed changes to the configuration of various elements (e.g., modify, delete, or reorder certain rules), and the new information will be sent to the analysis engine, which then performs a quick re-analysis to see if the (hypothetically) modified configuration information now conforms to the specification of the access policy. This feature can also be used to check planned changes in configuration for compliance quickly, before they are actually rolled out.

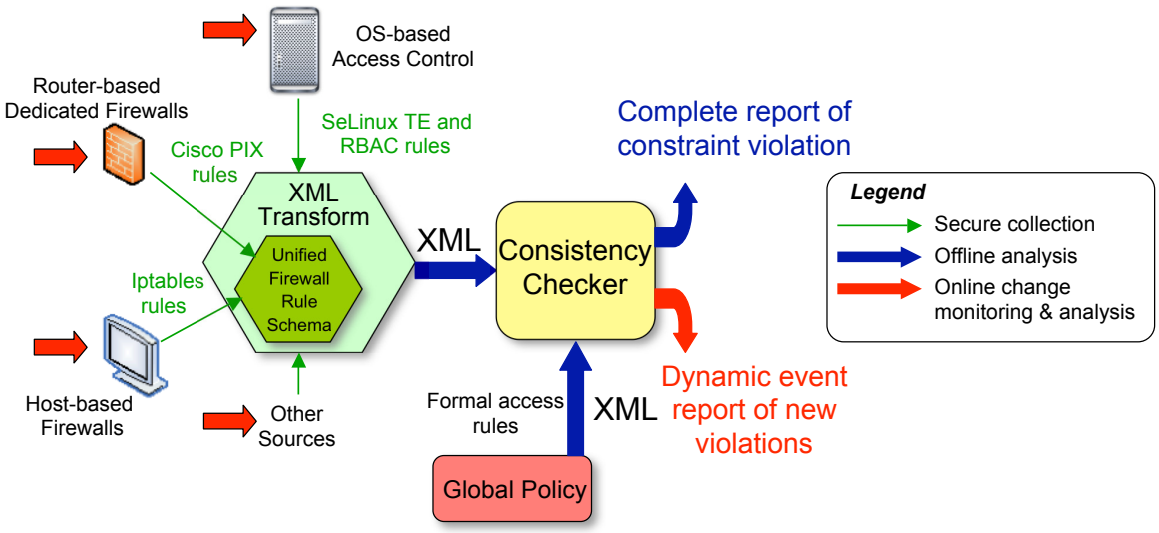

Figure 8.2: Operational Overview of the Analysis Engine

8.2 NetAPT Analysis Engine

Figure 8.2 gives an operational overview of the NetAPT analysis engine. The front-end supplies the analysis engine with information about the network topology and the parameters for establishing secure network connections with the access control elements of the topology. Examples of such parameters include keys for establishing VPN connections to Cisco PIX firewalls and a guest username and password for establishing SSH connections to Linux and SELinux hosts. Using the supplied information, the analysis engine can establish connections and obtain the relevant configuration information, hence capturing a snapshot of the access policy implementation. The configuration information can take many forms, depending on the mechanisms and devices being used to enforce access control. It may consist of custom rule descriptions for different kinds of firewalls, SELinux type and role transition policies, or Java Security Manager policies. NetAPT integrates policy rules from a large variety of such sources. We have developed a unified XML schema that captures the essence of the union of all the classes of access control mechanisms that one is likely to encounter in a modern IT network. The analysis engine includes modules that convert the policy rules (configuration information) from the different sources, with rules from each source in their own custom language, into XML conforming to our unified schema. That allows for easy extensibility, as support for new access control mechanisms and devices can be added by simply writing the translation modules for those devices.

In NetAPT's offline analysis mode, it is not necessary to tightly integrate the analysis engine with the system being analyzed; if needed, the analysis engine can be directed to read in the configuration snapshot that has been collected offline beforehand and placed locally on the file system. In the online analysis mode, the analysis engine either periodically checks with access control devices for any changes in configuration, or is specifically directed towards those changes by the user via the front-end.

The consistency checker forms the core of the analysis engine. It takes as input the XML representations of the collective configuration information from the various devices and of the formal specification of the access constraints provided by the front-end, and sets up an analysis based on the parameters supplied by the user (again, using the front-end). As described in Section 8.1, the output, in the form of the list of violations, is sent back to the front-end. In the online mode, if an intrusion detection and alert correlation system has been deployed on the network, the results may also be forwarded to that system in the form of an alert.

8.3 NetAPT Features

We now describe some additional NetAPT features, that have been particularly helpful in getting the tool adopted by a number of industry partners.

- *Authentication, Authorization, and Accounting (AAA)*: The first set of firewall rules provided to us contained access control lists (ACL) that applied to authentication and authorization. Of course, those authentication and authorization options exist in the Cisco PIX firewall rule specifications, but in our early development of NetAPT, we saw AAA as an area that we would address on an as-needed basis. The rules in those ACLs described the traffic that needs to be authenticated before normal filtering rules are applied. We also found commands for setting up the authentication mechanism (such as RADIUS, in which case the location of the RADIUS server was specified) and the incoming interface to the firewall to which the AAA-related ACL applied, among others. We consequently modified NetAPT to support
  - encoding of the AAA-related information into its universal XML schema for firewall rule-sets,

  - indication of global policy constraints that apply only to authenticated traffic, and
  - consideration of the above two aspects (rule-sets characterizing authenticated traffic and constraints applicable to such traffic) during the analysis.

  Later experience showed us that those features are apparently very common in industrial control environments and in corporate networks that support medium to high levels of security.

- *Object Groups*: Firewall configurations may support user definition of groups of objects, e.g., groups of networks, services, protocols, or users. The group definitions may then appear in a firewall rule, e.g., a network group might be identified as the set of destinations to which the rule applies. Our industrial partners used object groups extensively, as they are invaluable aids in describing large networks. We modified NetAPT to parse object group definitions (even recursive ones) and to provide a graphical interface to those definitions. We also enhanced NetAPT to allow a user to define NetAPT object groups to be used in the specification of global policy rules. For example, one object group can be used to encapsulate all the devices in the corporate network, and another to encapsulate all the devices in the control network; global policy rules that govern connections between the two networks can use the object group definitions in their specifications.

- *Topology Discovery*: In our initial design of NetAPT, we assumed that the user could specify his or her system's network topology. However, we discovered that our partners did not keep explicit topology information in a form that would allow for convenient communication of topological information to NetAPT. Furthermore, we were anxious to minimize our demands on partners' engineers' time in support of our efforts. We learned of the ANTFARM tool [51], communicated with its developers, and obtained it shortly thereafter, when it was released as open-source software. ANTFARM uses a relational database to coordinate disparate pieces of connectivity information. When a new connection is discovered, it is entered into the database, and an ANTFARM script is executed to forge any new connectivity relationships that are a consequence of the prior state of connectivity knowledge and the new knowledge. Working with ANTFARM and our partners, we discovered that our

partners needed to annotate some configuration information to help differentiate between networks, and that ANTFARM needed to parse that annotation. We have since made extensive customizations to the ANTFARM logic, again tailoring it to the technical requirements of topology discovery for our partners' networks. Issues related to topology discovery, verification, and validation have consumed a great deal of our attention. However, it has been worthwhile. In one notable session with a partner's network engineer, we found that NetAPT had discovered a path through the network that the engineer did not believe should be present. With NetAPT we were able to point out to him the configuration that allowed that path, and he agreed that the path must be present (and presumably made the configuration change to close the hole).

- *Network Islands*: One of our partners' configurations contained "switch" statements. Switch statements indicate that traffic leaving through a firewall is routed to another network, which is not necessarily adjacent to the firewall. Global policy statements may involve such flows, so we augmented NetAPT's topology discovery mechanisms to include the connectivity information specified by switch statements.

- *Command Line Processing*: One of our partners uses APT much less interactively than we envisioned. They keep track of changes made to firewall configurations using a tool called "rancid" (Really Awesome New Cisco config Differ) [52]. This tool maintains a CVS repository of the changes (diffs) made to the native firewall configurations and can be used to obtain the latest copy of the rule-sets for any firewall in the system. NetAPT uses the set of such rule-set files (in their native format, which uses the PIX/ASA command-set, all collected in a directory) as input for inferring topology and conversion to XML conforming to NetAPT's universal rule-set schema. This partner wants to use NetAPT as an off-line checker that would be run automatically every night against the current rule-set. Correspondingly, we augmented NetAPT so that it can be run entirely from a command line with explicitly named input and output files, and thus be run automatically through a "cron" script.

- *Enhanced Graphical Interface*: NetAPT's graphical display is written using Java libraries. Our initial display was written using the JGraph library [53], but we found that the size of the networks we were trying to display required use of a different tool. The NetAPT graphical engine was re-factored to use the Java Universal Network/Graph Framework (JUNG) [54] and the Model-View-Controller architectural pattern. The engine was augmented with features such as node clustering for easier viewing of large topologies, and zooming. It was also extended to provide multiple automatic layout schemes (easily accessible from a drop-down menu) for quickly drawing uncluttered representations of topology files. The users can then further arrange various elements to their satisfaction and save the topology back to the disk, such that the tool remembers all the changes made to the drawing. To further reduce clutter, the user can collapse various networks or collections of networks into a single element in the topology diagram, which can then be dragged around. Expanding such a collapsed node causes the constituent nodes to show up again, preserving their relative locations before the collapse.

- *Extended Range of Firewalls Supported*: NetAPT transforms firewall rules from specific firewall vendors and models of firewalls into a unified form, expressed in XML. The analysis engine works on this unified representation. Whenever a new firewall syntax is to be integrated into NetAPT, a "native-to-unified" translator needs to be augmented to support inclusion of the new device. Our partners had firewall models for which this extension was performed.

- *Global Policy Templates*: Our partners desired assistance in the development of global access policies. To facilitate that, we have built into NetAPT a library of global policy templates that express constraints based on best-practices recommendations such as those in NIST Special Report SP-800-82 [55]. The templates require a user to define object groups that identify sets of hosts and/or networks, and to substitute those group identifiers into the templates.

- *Global Policy Conflict Detection and Resolution*: A global policy is a collection of statements on flows that should be allowed and/or should not be allowed. The expression formalism is broad enough that conflicting rules can be concurrently specified. That is not

necessarily a bad thing; certain forms of conflict can be interpreted so as to give a clear and intuitive preference of one rule over another. Other conflicts cannot be so resolved, and indicate a need for the user to clarify the intent. We extended NetAPT to identify conflicts in global policies, and to point out the rules that could be resolved and those that need a user's attention.

CHAPTER 9

EXPERIMENTAL RESULTS

In this chapter, we describe how we demonstrated the efficacy of our analysis and topology inference algorithms, as implemented in NetAPT, through a number of experimental studies. We used several real-world-inspired test cases for our evaluations. Furthermore, to perform even more comprehensive studies, we have developed a benchmark suite of randomly generated network topologies and firewall rule-sets that are representative of the real-world network setups to which we had access.

9.1 Experimental Evaluation Using Test Cases

In this section, we discuss how we demonstrated the efficacy of NetAPT by using it to verify the access control policy implementation in a variety of settings. We began with test cases that were relatively small, such as the networking system typically found in a process control setting, and showed how the exhaustive analysis can be used to weed out misconfigurations of access control devices. We then demonstrated NetAPT's scalability through the use of statistical analysis for analyzing a much larger system that could not be handled by the exhaustive analysis. In all the experimental evaluations described below, NetAPT's analysis engine ran on an AMD AthlonXP-64 3700+ machine with 2GB of RAM.

9.1.1 Evaluation of Exhaustive Analysis

Figure 9.1 shows a testbed developed at the Sandia National Labs to represent a networking infrastructure at a typical Oil and Natural Gas (ONG) operator. The focus here is on the protection of the process control network. The network contains two dedicated Cisco PIX firewalls, each with

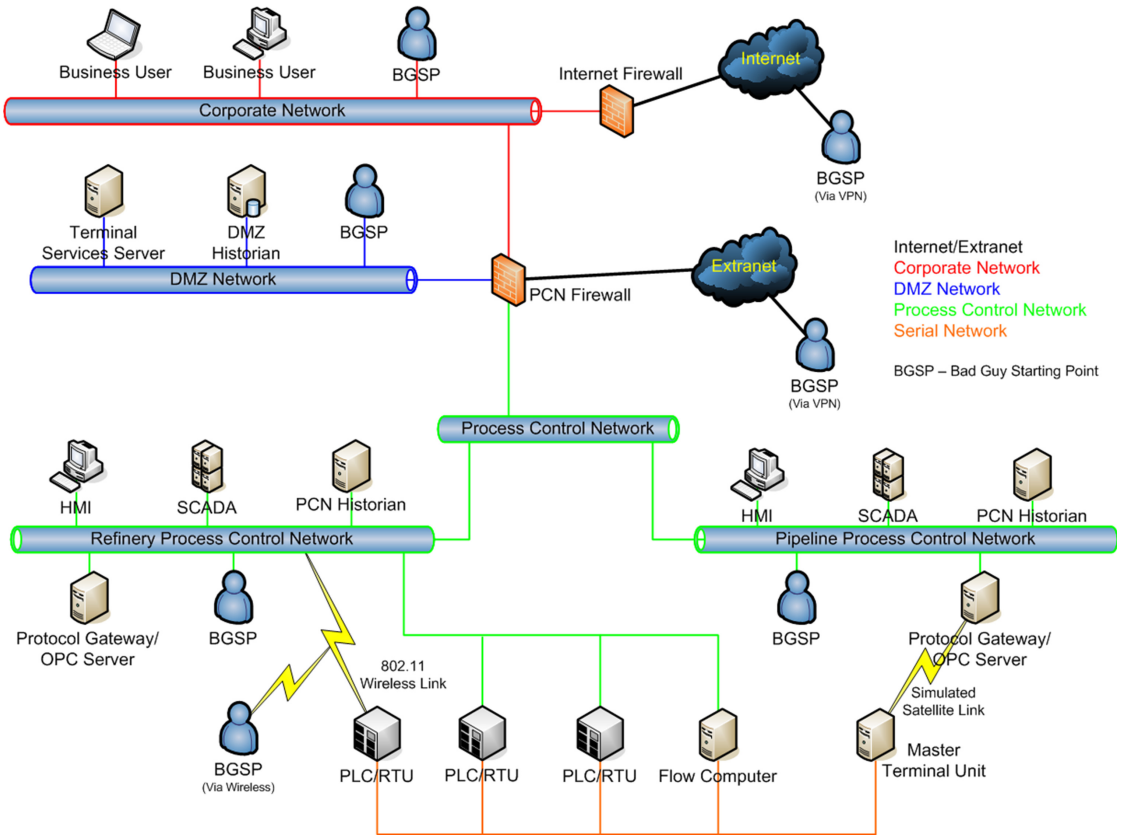

Figure 9.1: A Representative Process Control Testbed

15 rules. Out of the 17 hosts in the network, 5 have host-based firewalls (iptables in Linux). All the hosts run the stock versions of the Windows or Linux operating systems. The rule-sets for the various firewalls were populated based on the settings found at a number of actual ONG sites.

The global access constraints were defined, using NetAPT's graphical front-end, to emphasize the tightly controlled isolation of the process control network from the rest of the IT network. In particular, they specified that the hosts in the process control network could access each other; however, only one (the PCN Historian) could be accessed from outside the process control network. The only ways the PCN Historian could be accessed were via the DMZ Historian (by a "sysadmin" class user), or via a host in the corporate network (by a "manager" class user). NetAPT's graphical front-end facilitated easy specification of these constraints, which were subsequently translated into the underlying XML representation of the global policy.

NetAPT's analysis engine, using the information about the network topology provided by the front-end, automatically and securely captured the configuration snapshot of the system. An ex-

haustive analysis of the system was then performed. The analysis identified a total of 83 paths (sequences of firewall rules) in 10 seconds. The tool was then instructed to perform a post-analysis on the results to further pinpoint the root causes of the violations. The post-analysis, based on frequency of occurrence in the list of violations, identified four rules, two in each of the two Cisco firewalls acting in series, as the likely culprits. It further identified the attributes of the traffic that they were allowing to pass and also indicated the subset of the global access constraints that the traffic was violating. Using the graphical front-end, we were able to test hypothetical modifications to the highlighted rules and perform quick re-evaluation to discover the appropriate changes that would result in zero violations of the policy specification. As is evident from this experiment, NetAPT's exhaustive analysis functionality was very useful for a reasonably small network. A complete analysis of the testbed indicated that the rule-graph for the system had about 230,000 paths. Hence, the number of violations not only helps identify and correct misconfigurations, but also, when viewed in the context of the total number of ways traffic could traverse the system, can serve as a quantitative measure of the system's security posture.

We also used NetAPT to analyze the sample scripts used in the evaluation of the "Firewall Policy Advisor" by Al-Shaer and Hamed [17, 18, 19]. We captured all the misconfigurations in their sample scripts [19]. Note that NetAPT's functionality is a superset, since our analysis is not limited to the analysis of relations between just two firewalls, and we can check for inconsistencies (due to the use of multidimensional interval trees) as well as explicit policy violations.

To explore the scalability of the tool, we tested it on the example setup shown in Figure 2.1. This testbed represents an intrusion-tolerant publish-subscribe system developed as part of a DARPA-funded research effort. The system contains 1) over 40 hosts, of which 29 are running SELinux (each with 4 or more process domains) and all 40 have hardware NIC-based firewalls (with more than 20 rules per firewall), and 2) 8 Cisco PIX firewalls. As can be imagined, the configuration of the large number of access control elements in this system in adherence to the global security policy is an extremely complicated task.

The global access constraints were set to emphasize tightly controlled access to the "System Manager" group of machines (the top row of 4 machines in the figure). These machines could access one another in very specific ways (to run Byzantine fault-tolerance algorithms) and could be accessed by only a very small set of other machines, and only by processes from specific

domains on those machines. Again, NetAPT's graphical front-end and underlying XML schema were used for the specification of the global access constraints.

We then deliberately misconfigured a rule in the hardware NIC-based firewall for one of the system manager machines, allowing for traffic that would result in the violation of the global policy. The exhaustive analysis was able to identify the resulting 263 violations and pinpoint the root cause in about 140 seconds (focusing primarily on the firewall rules, and not on the host-based access control mechanisms). Note that the total number of paths in the rule-graph for this system is huge, which explains the time taken by the analysis.

We then made the problem more complex by introducing problems in two firewalls, a Cisco PIX dedicated firewall and the hardware NIC-based firewall for one of the system manager machines, such that the violation only occurred in the access decision sequences that included both rules. The exhaustive analysis, again limited to firewall rules, identified 155 violations and the two modified rules in about 200 seconds. However, when the host-based access control mechanisms were also included in the analysis, the exhaustive analysis could not complete, even after running for more than 3 hours, setting the stage for a demonstration of increased scalability provided by the statistical analysis.

9.1.2 Evaluation of Statistical Analysis

As described earlier, NetAPT's statistical analysis can provide a sample set of violations and an estimate of the remainder. For that purpose, we use the total number of violations as the quantitative estimate of the remainder. The biasing heuristic we use for guiding importance sampling is based on shortest distance, i.e., it assigns higher weight to those access decisions that would guide the traffic closest (in a network topology sense) to the hosts that are included in the specification of the global policy. The stopping criterion for the sampling was 500,000 samples or 5% relative error with 95% confidence, whichever was achieved first.

Statistical analysis on the Figure 2.1 testbed for the one-rule misconfiguration example described above resulted in an answer in under 10 seconds (with the relative error convergence being the stopping criterion). The analysis estimated 255 violations, which is within 3% of the exact answer obtained from exhaustive analysis.

For the example with two misconfigured rules, the statistical analysis obtained an answer within 4% of the exact answer in about 10 seconds when only firewall rules were being analyzed, and in about 25 seconds when host-based access control mechanisms were also being modeled.

We tested another example, in which, in addition to the two misconfigured rules, we also introduced a misconfiguration in the SELinux policy for the host with the misconfigured host-based firewall, such that the global policy was now violated only when all three access control points were included. Again, the exhaustive analysis could not produce the complete list of violations after running for more than 3 hours, but the statistical analysis was able to provide an estimate with a 10% confidence interval (where the number of samples analyzed was the stopping criterion) in about 1 minute. Additional accuracy required a disproportionate increase in the time required for the analysis. To obtain an estimate with a 5% confidence interval, it was necessary to run the analysis for about 5 minutes.

Hence, we can see that statistical analysis allows NetAPT to analyze fairly complex systems within reasonably short periods of time. The tool's performance can likely be improved further with the use of better biasing heuristics, which is an avenue that we will continue to explore in the future.

9.1.3 Field Testing at a Multi-Site PCS

NetAPT aims to facilitate government-mandated compliance audits. We field-tested NetAPT at a large utility with a networked process control system that was spread over multiple geographical sites. Virtual Private Networks (VPNs) and direct fiberoptic connections were used to connect the different sites. The network had more than 90 firewalls, with a collective total of more than 8000 rules. The network contained more than 3000 hosts.

NetAPT was given access to the rule-sets of all the firewalls, which were maintained in a central repository using the *rancid* tool.

NetAPT's topology generation was able to generate a topology diagram for the underlying network in about 50 minutes. Figure 9.2 shows an anonymized version of a part of the network topology inferred. The diagram was generated using NetAPT's graphical front-end. Briefly, the red circles indicate firewalls, the green circles indicate routers or switches, and the other circles in-

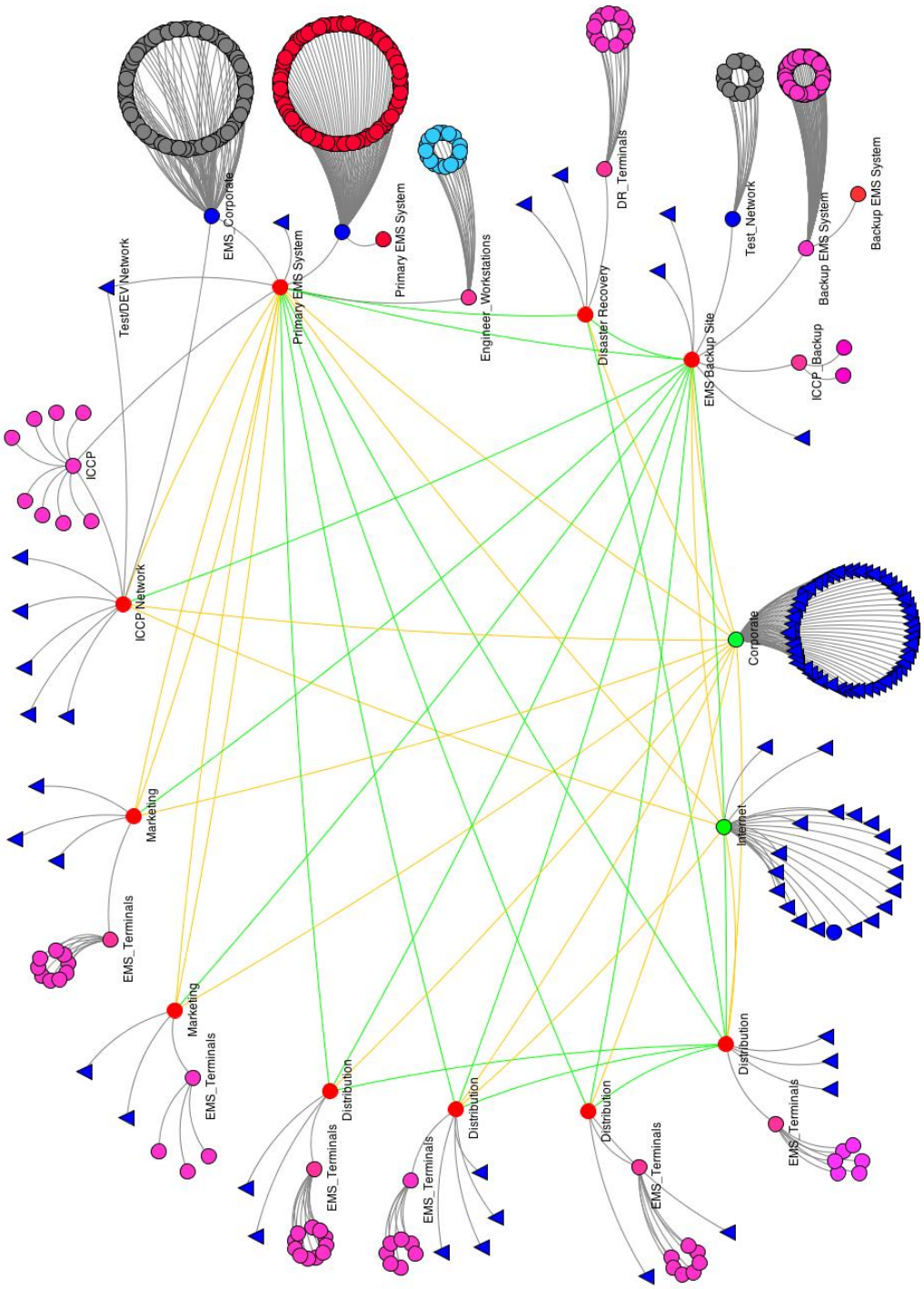

Figure 9.2: A Network Topology Inferred by NetAPT (Anonymized)

dicate hosts. The blue triangles represent sub-networks, each containing multiple hosts, that have been collapsed in the diagram to avoid visual clutter. The green arcs represent VPN tunnels, and the yellow arcs represent direct fiberoptic connections. The system administrators and network engineers at our industrial partner verified that the inferred network was correct and captured all the hosts and sub-networks.

As described in Chapter 8, NetAPT includes parameterized policy templates that encode various best-practices recommendations and compliance requirements. For the field test, we chose the policy template that encoded the NERC CIP 005 compliance requirements [56]. We were able to customize the policy template for the inferred network topology in a couple of hours using the policy specification component of NetAPT.

An exhaustive analysis of the inferred topology for compliance with the NERC CIP 005 policy was completed in 55 seconds, during which multiple exceptions were identified. The network administrators were able to visually highlight and annotate each exception identified. There were zero false alarms; each of those exceptions was attributed to either a problem in the firewall configurations or an expansion of the policy.

Ultimately, NetAPT helped produce comprehensive and highly visual reports that helped our industrial partner prove that their network was in compliance with the NERC CIP standards. They were able to correct some problems with their configurations, and the entire process took considerably less time than the usual two-week audit process, the latter being significantly less comprehensive, as it was done on selected parts of the network by hand.

9.2 Automatic Generation of Representative Networks

So far, we have demonstrated the efficacy of our techniques on several test cases. However, there is a need for a more comprehensive experimental study of the performance and scalability of our analysis and topology inference algorithms, and the completeness of the inference algorithms.

We have collaborated with multiple industrial partners during our research, and have obtained access to a number of real-world network topologies and firewall rule-sets through those collaborations. The majority of those networks are SCADA networks representing process control systems

and the associated enterprise sub-networks. However, for several reasons, those data sets are not sufficient for comprehensive experimental studies.

The real-world network topologies are limited to specific data points and do not immediately allow us to study how our algorithms scale when the study parameters are varied in particular ways. Second, we have observed that while networks share certain essential characteristics, the larger networks in our data set were not obtained simply by siloing and replicating the smaller networks in the data set. Hence, a trivial scaling technique was not possible.

We were also under nondisclosure agreements with our industrial partners that severely limited the extent to which results of analysis of their networks could be publicly discussed. Those agreements also allowed us to use the data sets only with explicitly permitted techniques, ruling out new algorithms and mechanisms we may develop in the future. In addition, we wanted to have a realistic benchmark that could be of use to other researchers.

For the reasons indicated above, we decided to develop an algorithm for generating random network topologies and firewall rule-sets that captured the essential characteristics of a set of training topologies and corresponding rule-sets. We generated the benchmark topologies by varying parameters such as numbers of sub-networks, access control devices (mostly firewalls), hosts and applications deployed, and the rule-set size and complexity.

9.2.1 Related Work and Background

There has been considerable work on the generation of random networks. Those efforts have been applied to multiple domains, including networks of computer systems, social networks, and networks characterizing spread of infections. They are not directly applicable to our goals as they have been too generic, and the generation models involved were not based on the learned structural and flow (traffic) characteristics of a set of training networks. Still, several ideas that emerged from those efforts motivated our algorithms.

In the Small World Experiment, Milgram sent a package to several randomly chosen people in Wichita, Kansas, with instructions asking them to forward an enclosed letter to a target person, identified by name and occupation, in Sharon, Massachusetts. The subjects were told that they could mail the letter directly to the target person only if they knew him personally; otherwise they

were instructed to send it, and the same instructions, to a relative or friend they thought would be more likely to know the target person.

Many of the letters were never delivered, but for the ones that were the average path length—the number of times the letters were forwarded—was about six. This result was taken to confirm previous observations (and speculations) that the typical distance between any two people in a social network is about "six degrees of separation."

The conclusion was surprising, because most people expect social networks to be localized—people tend to live near their friends—and in a graph with local connections, path lengths tend to increase in proportion to geographical distance.

In 1998, Duncan Watts and Steven Strogatz published a paper in *Nature*, "Collective Dynamics of 'Small-World' Networks," [57] that proposed an explanation for the small-world phenomenon. They started with two kinds of graph that were well-understood: random graphs and regular graphs. They looked at two properties of these graphs: clustering and path length.

Clustering is a measure of the "cliquishness" of the graph. In a graph, a *clique* is a subset of nodes that are all connected to each other; in a social network, a clique is a set of friends who all know each other. Watts and Strogatz defined a clustering coefficient that quantifies the likelihood that two nodes that are connected to the same node are also connected to each other.

Path length is a measure of the average distance between two nodes, which corresponds to the degrees of separation in a social network.

Their initial result was what might be expected: regular graphs have high clustering and high path lengths; random graphs with the same size tend to have low clustering and low path lengths. So neither of these is a good model of social networks, which seem to combine high clustering with short path lengths.

Watts and Strogatz's goal was to create a *generative model* of a social network. A generative model tries to explain a phenomenon by modeling the process that builds or leads to the phenomenon. In this case they proposed a process for building small-world graphs:

1. Start with a regular graph with n nodes and degree k. (Watts and Strogatz start with a ring

lattice, which is a kind of regular graph. One could replicate their experiment or instead try a graph that is regular but not a ring lattice.)

2. Choose a subset of the edges in the graph and "rewire" them by replacing them with random edges. Again, one could replicate the procedure described in the paper or experiment with alternatives.

The proportion of edges that are rewired is a parameter, p, that controls how random the graph is. With $p = 0$, the graph is regular; with $p = 1$, it is random.

Watts and Strogatz found that small values of p yield graphs with high clustering, like a regular graph, and low path lengths, like a random graph.

Zipf's law [58] describes a relationship between the frequencies and ranks of words in natural languages. The "frequency" of a word is the number of times it appears in a body of work. The "rank" of a word is its position in a list of words sorted by frequency: the most common word has rank 1, the second most common has rank 2, and so forth.

Specifically, Zipf's law predicts that the frequency, f, of the word with rank r is:

$$f = cr^{-s}$$

where s and c are parameters that depend on the language and the text.

If we take the logarithm of both sides of this equation, we get:

$$\log f = \log c - s \log r$$

So if we plot $\log f$ versus $\log r$, we should get a straight line with slope $-s$ and intercept $\log c$.

The Pareto distribution [59] is named after the economist Vilfredo Pareto, who used it to describe the distribution of wealth. Since then, people have used it to describe phenomena in the natural and social sciences, including sizes of cities and towns, sand particles and meteorites, forest fires, and earthquakes.

The Pareto distribution is characterized by a CDF with the following form:

$$CDF(x) = 1 - \left(\frac{x}{x_m}\right)^{-\alpha}$$

The parameters x_m and α determine the location and shape of the distribution. x_m is the minimum possible quantity.

Values from a Pareto distribution often have these properties:

***Long tail*:** Pareto distributions contain many small values and a few very large ones.

***80/20 rule*:** The large values in a Pareto distribution are so large that they make up a disproportionate share of the total. In the context of wealth, the 80/20 rule says that 20% of the people own 80% of the wealth.

Scale-free: Short-tailed distributions are centered around a typical size, which is called a "scale." For example, the great majority of adult humans are between 100 and 200 cm in height, so we could say that the scale of human height is a few hundred centimeters. But for long-tailed distributions, there is no similar range (bounded by a factor of two) that contains the bulk of the distribution. So we say that these distributions are "scale-free."

To get a sense of the difference between the Pareto and Gaussian distributions, imagine what the world would be like if the distribution of human height were Pareto.

The equation for the CCDF is:

$$y = 1 - CDF(x) \sim \left(\frac{x}{x_m}\right)^{-\alpha}$$

Taking the log of both sides yields:

$$\log y \sim -\alpha(\log x - \log x_m)$$

So if one plots $\log y$ versus $\log x$, the result should look like a straight line with slope $-\alpha$ and intercept $\alpha \log x_m$.

One can measure the degree (number of connections) of each node and compute $P(k)$, the probability that a vertex has degree k; then one can plot $P(k)$ versus k on a log-log scale. The tail of the plot fits a straight line, so one can conclude that it obeys a *power law*; that is, as k gets large, $P(k)$ is asymptotic to $k^{-\gamma}$, where γ is a parameter that determines the rate of decay.

Barabási and Albert [60] also propose a model that generates random graphs with the same property. The essential features of the model, which distinguish it from the Erdos-Renyi model and the Watts-Strogatz model, are:

Growth: Instead of starting with a fixed number of vertices, Barabasi and Albert start with a small graph and add vertices gradually.

Preferential attachment: When a new edge is created, it is more likely to connect to a vertex that already has a large number of edges. This "rich get richer" effect is characteristic of the growth patterns of some real-world networks.

Finally, one can show that graphs generated by this model have a distribution of degrees that obeys a power law. Graphs that have this property are sometimes called *scale-free networks*. That name can be confusing, because it is the distribution of degrees that is scale-free, not the network.

In order to maximize confusion, distributions that obey the power law are sometimes called *scaling distributions* because they are invariant under a change of scale. That means that if one changes the units in which the quantities are expressed, the slope parameter, γ, doesn't change.

At this point, we have seen three phenomena that yield a straight line on a log-log plot:

- Zipf plot: Frequency as a function of rank.
- Pareto CCDF: The complementary CDF of a Pareto distribution.
- Power law plot: A histogram of frequencies.

The similarity in these plots is not a coincidence; these visual tests are closely related.

Starting with a power-law distribution, we have:

$$P(k) \sim k^{-\gamma}$$

If we choose a random node in a scale-free network, $P(k)$ is the probability that its degree equals k.

The cumulative distribution function, $CDF(k)$, is the probability that the degree is less than or equal to k, so we can get that by summation:

$$CDF(k) = \sum_{i=0}^{k} P(i)$$

For large values of k, we can approximate the summation with an integral:

$$\sum_{i=0}^{k} i^{-\gamma} \sim \int_{i=0}^{k} i^{-\gamma} = \frac{1}{\gamma - 1}(1 - k^{-\gamma+1})$$

To make this a proper CDF, we could normalize it so that it goes to 1 as k goes to infinity, but that's not necessary, because all we need to know is:

$$CDF(k) \sim 1 - k^{-\gamma+1}$$

That shows that the distribution of k is asymptotic to a Pareto distribution with $\alpha = \gamma - 1$.

$$f = cr^{-s}$$

where f is the frequency of the word with rank r. Inverting this relationship yields:

$$r = (f/c)^{-1/s}$$

Now, subtracting 1 and dividing through by the number of different words, n, we get

$$\frac{r-1}{n} = \frac{(f/c)^{-1/s}}{n} - \frac{1}{n}$$

,

which is only interesting because if r is the rank of a word, then $(r-1)/n$ is the fraction of words with lower ranks, which is the fraction of words with higher frequency, which is the CCDF of the distribution of frequencies:

$$CCDF(x) = \frac{(f/c)^{-1/s}}{n} - \frac{1}{n}$$

To characterize the asymptotic behavior for large n, we can ignore c and $1/n$, which yields:

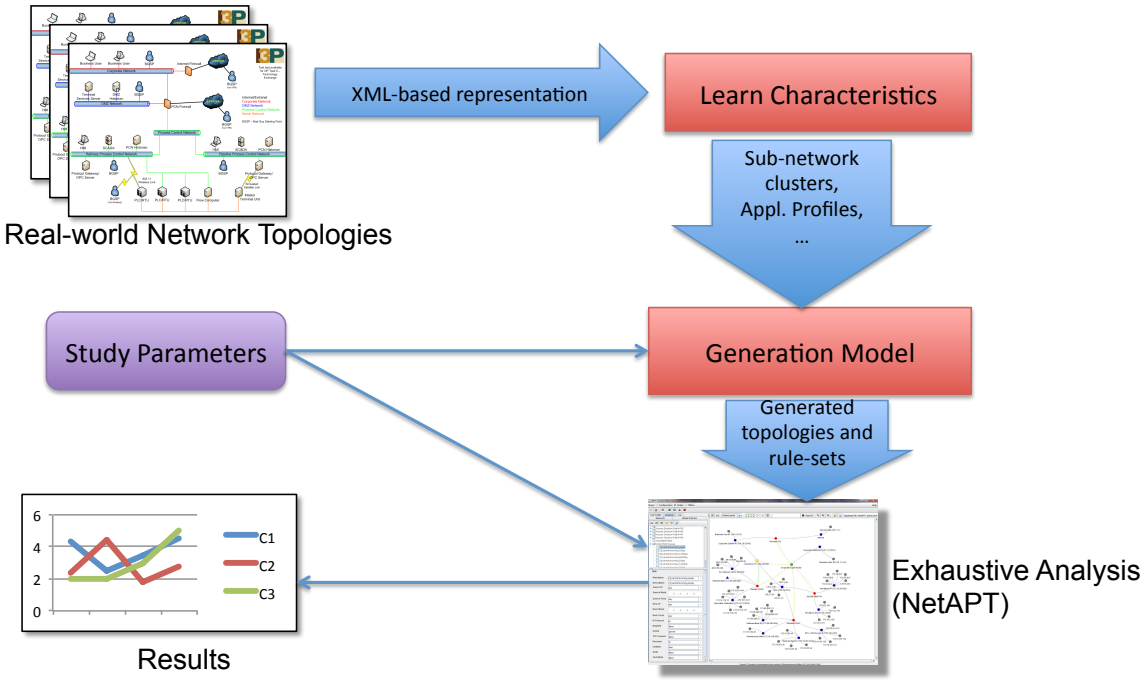

Figure 9.3: Evaluating NetAPT Using Randomly Generated Networks

$$CCDF(x) \sim f^{-1/s}$$

That shows that if a set of words obeys Zipf's law, the distribution of their frequencies is asymptotic to a Pareto distribution with $\alpha = 1/s$.

So the three visual tests are mathematically equivalent; a dataset that passes one test will pass all three. But as a practical matter, the power law plot is noisier than the other two, because it is the derivative of the CCDF.

The Zipf and CCDF plots are more robust, but Zipf's law is only applicable to discrete data (like words), not continuous quantities. CCDF plots work with both. For these reasons—robustness and generality—we used CCDFs where applicable in our own algorithms.

9.2.2 Overall Approach

Our overall approach to comprehensive experimental evaluation of NetAPT is shown in Figure 9.3. We made the following observations from the empirical data sets to which we had access:

- Networks are organized into sub-networks, with the most atomic of the sub-networks representing hosts on a single local area network.
- Sub-networks that collectively form the overall network can be classified into a small set of classes or types. For example, the networks in Figure 9.1 can be classified into the types: "corporate network," "DMZ," and "control network."
- The networks support a finite set of applications, and the firewall rule-sets determine the flow of traffic to and from the deployed applications.

Based on the above observations, we first learn the characteristics of the training topologies. We capture the network topology structure by using an algorithm to classify the sub-networks in the training topologies into various types or classes, and identifying the ways in which the sub-networks of various types connect with each other. We capture the connectivity and network traffic flows implied by the firewall rule-sets by developing traffic profiles for each deployed application or service.

The generation algorithm then uses the probability distributions stored and associated with the sub-network classes and application profiles to generate samples for new networks and rule-sets.

9.2.3 Learning Characteristics

Characterizing Network Topology Structure

As stated earlier, the first step in learning the structural characteristics of a set of training topologies to classify the types of sub-networks present in those topologies. We achieve that by performing an automatic hierarchical classification of networks.

We use a variation of *agglomerative clustering* to obtain our classification. The algorithm starts with individual sub-networks in our data set and successively merges similar sub-networks into clusters. Clusters then represent classes or types of network with varying degrees of resolution or abstraction. Figure 9.4 shows an example partial dendrogram resulting from the agglomerative clustering of the network shown in Figure 9.1. Note that the labels shown in Figure 9.4 have been added for ease of exposition, and are not automatically generated by our algorithm.

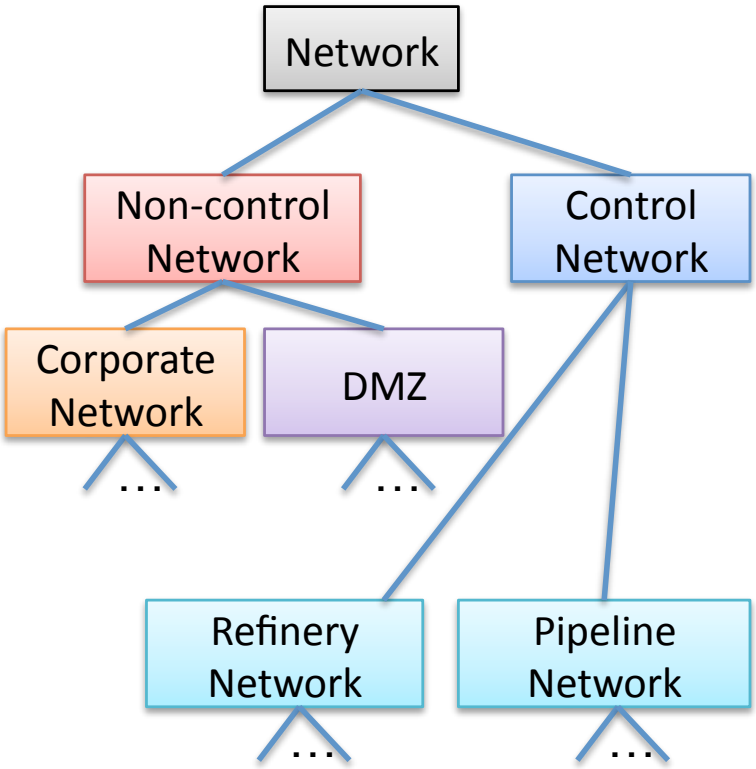

Figure 9.4: A Example (Partial) Classification of Network Types

Figure 9.5 shows the information stored with each node in the dendrogram. In particular, we store:

- *List of services hosted or provided* by a sub-network of the type represented by the node, along with the likelihood of each service being hosted.
- *List of external services accessed* by a sub-network of the type represented by the node, along with the likelihood of each service being accessed.
- *Relative size* of a sub-network of the type represented by the node. It is the likely ratio of the number of hosts in a sub-network of this type to the total number of hosts in the overall network.
- *Relative frequency* of the occurrence of a sub-network of the type represented by the node. It is the likely ratio of the number of sub-networks of this type to the total number of sub-networks that form the overall network.

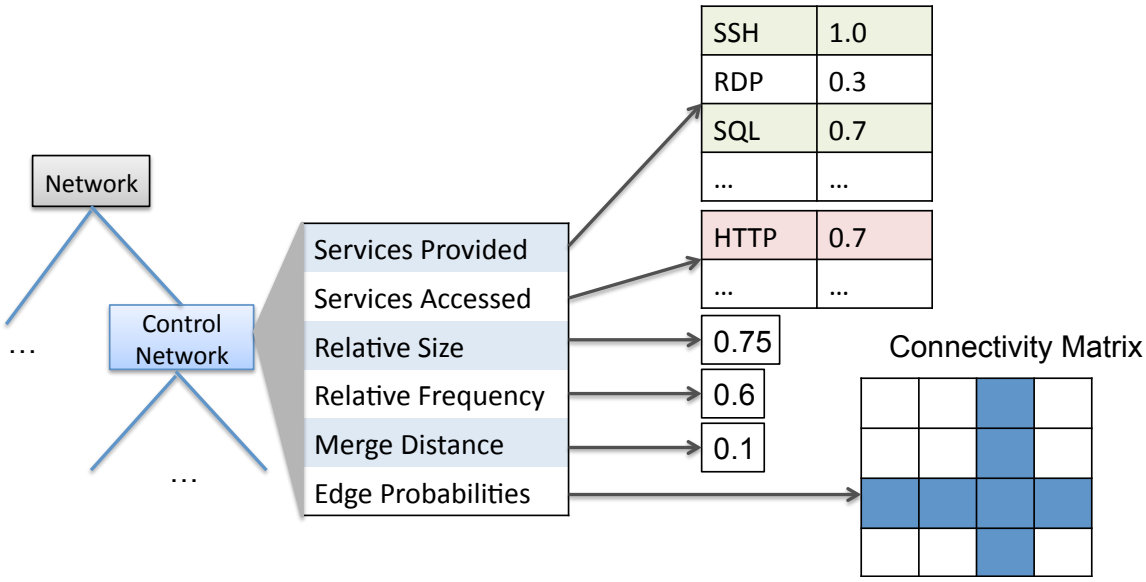

Figure 9.5: Information Stored in a Cluster Node

- *Edge probabilities*: for each node in the dendrogram, the probability that a sub-network of the type represented by that node is connected to a sub-network of the type represented by this node. The edge probabilities for all the nodes are stored in a single adjacency-matrix-like representation, that we call the *connectivity matrix*.

- *Merge distance* between the two child nodes of the current node, whose merge (or clustering) produced this node. We describe the metrics and calculations for the merge distance below.

To facilitate clustering of nodes through merges, we assign a *traffic profile vector* to each node.

Definition 12. *Let $S_{in} = \{s_{in}^1, \ldots, s_{in}^n\}$ be the set of all possible services provided by a sub-network in our training data set. Let $S_{out} = \{s_{out}^1, \ldots, s_{out}^m\}$ be the set of all possible services accessed by a sub-network in our training data set. We consider an $(n+m)$ dimensional vector space that has a dimension associated with members of S_{in} and S_{out}. In this vector space, a given cluster node is assigned the Cartesian coordinates representing the likelihood of the corresponding service being provided or accessed by a sub-network of the type represented by the cluster node. Those $(n+m)$ Cartesian coordinates define a cluster node's traffic profile vector.*

We use the traffic profile vectors to determine similarity of nodes and use that information to identify the nodes that can be clustered.

The clustering algorithm has the following steps:

1. We identify the individual sub-networks from the XML-based representation of the training network topologies. We can do so by enumerating all the `Network` elements in the DTD shown in Figure 4.3. Those sub-networks form the leaves or the atomic clusters of our hierarchical classification dendrogram. Each sub-network is initially labeled with a set of keywords. That set is a union of any user-specified description, interface name or descriptions for the sub-network in the firewalls connected to the sub-network, and the names of the object-group(s) of which sub-network may be a part.

2. We perform an initial set of merges among the clusters identified in Step 1. We merge the clusters that represent sub-networks with similar labels. We define certain label strings to be equivalent to further aid the merge process (e.g., DMZ is equivalent, to "Demilitarized Zone"). We set the *merge distance* field in the merged node to 0.

3. We merge the two clusters with the least traffic profile distance, based on the profiles of the incoming and outgoing traffic. The traffic profile distance is the Euclidean distance between the traffic profile vectors representing the two nodes. We store the traffic profile distance between the two merging nodes in the *merge distance* field of the merged node. We update the other fields in the merged node with appropriate weighted means of the corresponding values for the merging nodes. We also expand and update the connectivity matrix with rows and columns associated with the newly formed merged node.

4. We repeat step 3 above until there is only one node left.

At the end of the above steps, we have a dendrogram representing a hierarchical classification of sub-network types.

Characterizing Traffic Patterns

We develop *application profiles* to capture the essential characteristics of the traffic patterns observed in the training data set.

We begin by identifying the different applications being used on the networks in the training data set. We construct an initial set of applications, adding a member for each unique destination port on which a service receives traffic. We then compact this set by merging together members with the same or similar object group names, aliases, identifiers, and descriptions. We allow for further merging based on user input. During the merge operations, we keep track of the set of destination ports on which each application represented by the merged member can receive traffic. The final compacted set, $\mathscr{A}$, represents the applications to be profiled. Some examples that we identified from our real-world topologies include remote access using the Remote Desktop Protocol (RDP) and data historian services using SQL. In all the tests with the real-world training data, we have found that $|\mathscr{A}|$ is usually around 12, and never exceeds 20.

For each application in $\mathscr{A}$, we look at paths through the network that end up accessing that application, and we associate with that application a set of the unique path lengths encountered, where each member of the set also stores the number of times that a path of that length was seen.

We also calculate and store the probabilities of different combinations of services being deployed on the same host. While this is a set of size $O(2^{|\mathscr{A}|})$, that size is not likely to be a problem, as we use a sparse data structure (storing only the nonzero probabilities), and $|\mathscr{A}| \leq 20$ in our empirical observations.

9.2.4 Generation Algorithm

Generating Topology

We now describe the generation algorithm (Generation algorithms are also known referred to as *generative models* in the literature.) that we use to generate random network topologies.

As noted in Section 9.2.1, the prior efforts, such as the Watts-Strogatz model [57] and the Barabási-Albert model [60] for scale-free networks, are too generic, and do not capture the essential characteristics of our real-world training data set.

The generation algorithm requires three sets of input:

- List of services or applications to be deployed.

- Number of sub-networks (separated by layer 3 devices and belonging to a single LAN) that form the overall network.

- Overall size of the network as characterized by the number of hosts.

Given the above input, the first step in the generation algorithm is to select the sub-network classes that would form the overall network topology.

We consider the dendrogram representing the hierarchical classification of sub-network types (e.g., Figure 9.4). We initialize the "distance threshold," d, to be the *merge distance* stored in the root node of the dendrogram.

We know from the description of the agglomerative clustering algorithm above that the merge distance stored at the root node is the largest of all merge distances stored at the nodes of the dendrogram. We then successively reduce the distance threshold, selecting the next largest merge distance in each step. At each step i, we consider the nodes whose merge distances are greater than or equal to the current value of d. Let those nodes be represented by N_i. Together, the nodes in N_i form G_i, a sub-tree rooted at the root of the dendrogram.

We continue to decrease d until either the number of leaf nodes in G_i is greater than or equal to the number of sub-networks provided as input to the algorithm, or we exhaust the dendrogram. Let G_f denote the sub-tree in that final step.

We randomly select a set of nodes in G_f that collectively cover the input set of services deployed to indicate N_C, the set of sub-network classes that will be represented in the generated topology. The probability that a node is selected is a function of the value of the node's *relative frequency* field and the node's "resolution" (graph-theoretic distance from the root node of the dendrogram), with nodes with higher frequencies and higher resolutions being more likely to be selected. If a node is selected for N_C, none of its descendant nodes in the hierarchy dendrogram are selected for N_C.

Once we have selected the classes (or types) of sub-networks to be represented, we instantiate the actual sub-networks. We use the normalized relative frequency of occurrence of the sub-network classes in N_C to determine the number of sub-networks of each type instantiated. We use N_I to denote the set of instantiated sub-networks.

The next step in the generating algorithm is to assign applications (services deployed and accessed) to each instantiated sub-network. For each sub-network in N_I, we initially assign services by sampling from the probabilities of the services hosted and accessed, stored at the node cluster (sub-network type) of which the sub-network is an instance. After that initial assignment of services, we look for applications provided as input to the generating algorithm that were not deployed in the initial assignment. We attempt to assign each of those applications to one or more sub-networks in N_I based on the relevant probabilities stored with the corresponding node cluster (sub-network type).

We connect the instantiated sub-networks using an iterative algorithm that adds edges to the topology being generated based on the various probability distributions associated with sub-network types, either explicitly or implicitly.

- We sample from the *connectivity matrix* associated with the hierarchical classification of the sub-network types to add an initial set of edges. In other words, we use the likelihood of single-hop connection between node clusters to determine when to add edges between instantiated sub-networks of those types.

- For each deployed application, starting with the application for which the path lengths stored in its profile have the lowest mode:

  - We sample a path length based on the discrete distribution stored in the application's profile, say l.

  - We add an edge from a sub-network that accesses the application to a sub-network at the distance of $(l-1)$ from a sub-network that hosts the application. If multiple qualifying sub-networks that host the application are present, we pick the one that has the highest current degree. This introduces certain scale-free characteristics into the network being generated.

We now place firewalls on certain communication paths (edges) in our network. Consider a clique with n nodes (sub-networks), that has m edges between to sub-networks in the clique and sub-networks not in the clique. We replace such a clique in our network topology with a firewall with $(n+m)$ interfaces, such that the firewall has an interface connected to each sub-network in

the clique, and an interface connected to each of the non-clique sub-networks that were connected to the clique.

We populate the instantiated sub-networks with hosts based on information about the relative size associated with the corresponding sub-network type and the overall size of the network requested. We deploy the services that are assigned to an instantiated sub-network on individual hosts in the sub-network by sampling from the stored probabilities of application combinations.

Generating Rule-Sets

Each time we add a path in the network topology during the algorithm described above, we store information about the relevant traffic in a 4-D interval tree (similar to that used for representing a TAS). After the algorithm has generated the network topology, we combine all the 4-D interval trees into a single 4-D interval tree using efficient operations described in Section 4.2.1. We use an extended version of the algorithms described in [43, 44] to instantiate a minimal set of firewall rules for the individual firewalls that collectively produce precisely the traffic represented by the combined 4-D interval tree.

If the generation parameters included the average rule-set size, we add additional "accept" rules about the services explicitly mentioned in the input to the generation algorithm, and add "accept" or "deny" rules for services not explicitly mentioned in the input parameters. Addition of these rules does not affect access to the explicitly mentioned services. We continue to add such rules until the requirements about the average rule-set size are met. Our algorithm biases the addition of new rules to firewalls whose rule-set sizes are below the current average.

9.2.5 Concluding Remarks

We have described a framework for learning the essential characteristics of a set of training topologies and rule-sets and using those characteristics to generate "similar" random network topologies and accompanying firewall rule-sets. We are currently working on generating benchmarks that can be used for a comprehensive study of the algorithms implemented in NetAPT. We now describe some of the ways such a benchmark can be constructed.

We can study the performance of the exhaustive analysis algorithm on randomly generated networks that satisfy variety of criteria. For example, some of those criteria (or study parameters given to the generation algorithm) can be:

- Network size goes from 2 to 1000 sub-networks, with fixed number of total hosts, while using the minimal rule-set. The number of total hosts can be varied from 1000 to 10000.
- Network size goes from 2 to 1000 sub-networks, with proportional number of total hosts (5x, 10x), while using the minimal rule-set.
- Vary the number of applications deployed for a fixed network size and minimal rule-set.
- Vary the average size of a rule-set for a fixed network size and a fixed set of applications deployed.

All of the above examples would require generation of multiple sample networks for each data-point and the analysis would have to be run until sufficiently narrow confidence intervals are reached on the performance metrics being estimated.

We can study the worst-case performance of our analysis algorithms by using a "deny-all" global policy. The exhaustive analysis against such a policy generates the connectivity map for the network system being analyzed. To further refine, we can indicate the classes of misconfigurations that are to be introduced, using the error model described in Chapter 2, along with the frequency of each class of errors. We can then check against an automatically generated policy that only permits the traffic destined for the deployed applications and denies everything else.

CHAPTER 10

CONCLUSIONS

As networked systems continue to grow in scale and complexity, and are being used in increasingly critical systems, it becomes important to make sure that they are protected from cyber attacks. In particular, it is essential that the access to the critical resources on the network is carefully controlled such that those resources are not exposed to attackers. Deployment of a large number of distributed and layered access control mechanisms, such as firewalls, is the staple solution to the problem of enforcing security policy. Hence, it is very important to ensure that all the access control mechanisms work collectively in harmony, and that their complex interactions do not mask subtle errors, introducing security vulnerabilities.

To provide system administrators and auditors with tools to help identify security problems that would otherwise be difficult to detect without a considerable time investment, we have presented a formal framework for the analysis of security configurations. We have presented the details of the formalisms, as well as various data structures and algorithms used for a space and time efficient implementation of the proposed framework. We have demonstrated the efficacy of our algorithms through analytic complexity analyses and some preliminary experimental results. We have also provided a formal description of the preliminary setup for performing statistical analysis of very large and complex systems. In this chapter, we briefly review the work we have presented in this dissertation and describe potential avenues for the expansion of our work, before concluding with some final remarks.

10.1 Contribution Review

We presented formalisms for representing various aspects of the problem, namely the global policy and the network system, where the latter is characterized by the network topology and the rule-sets

of all the firewalls on the network. We provided an XML-based policy specification language, that was proven to be at least as expressive as LTL. Our policy specification language strikes a balance between human-readability (that often leads to verbosity) and a sound mathematical basis (that allows formal reasoning). We provided a unified XML-based schema for representing rule-sets from a variety of sources, including multiple models and makes of firewalls and operating system-based mechanisms. We also provided a feature-rich XML-based representation of the network topology.

Once we were able to characterize the inputs to our analysis algorithms, using the formalisms described above, we presented the multilayered rule-graph, a specialized data structure that we use to facilitate efficient exploration of paths through the network while checking for policy compliance and inconsistencies.

We presented a complete classification of possible errors that can be found in firewall rule-sets due internal inconsistencies and violations of a user-specified global access policy. We next presented an efficient algorithms for a comprehensive exhaustive analysis of the policy implemented in a networked system (firewall rule-sets) for compliance with a specification of the global access policy. That algorithm enumerates all the internal inconsistencies found in rule-sets, lists all the paths through the network that result in violation of some aspect of the global policy specification, and for those violating paths, identifies the small set of likely root causes (rules that were misconfigured, resulting in that paths that violate the global policy). We explored ways to further optimize the performance for the exhaustive analysis algorithm and provide analytic evaluation in the form of a space and time complexity analysis.

Exhaustive analysis has its limitations as the system being analyzed scale and become more complex. That is especially true if we include operating system-based access control mechanisms. To handle such cases, we introduced an importance sampling-based statistical analysis algorithm. We provided a formal description of the algorithm and included some assurances about the variance reduction achieved by the algorithm. The statistical analysis either results in a sample set of violations with a quantitation estimate of the remainder, or a confidence measure of there being no violations.

During our interaction with several industry partners over the course of our research, we observed the lack of detailed maps of the network topology, which is an essential input for the analy-

sis algorithms. To alleviate that issue, we presented an algorithm to automatically infer the network topology from the configurations of the firewalls and other layer 3 devices in the network.

We presented a brief description of the Network Access Policy Tool that implements our framework. We then applied NetAPT to various test cases obtained through industrial collaboration, and demonstrated that our methods were of great use to real system administrators and auditors in identifying problems in the configurations of their access control devices. Finally, we presented a framework for learning the essential characteristics of our real-world network topology data sets, and using those characteristics to create a benchmark suite of automatically generated representative random networks and firewall rule-sets of varying sizes and complexity. We indicated how the benchmark can be used for a comprehensive study of the performance and correctness of our algorithms.

10.2 Future Avenues

10.2.1 Fast Incremental Change Analysis

This task entails the design and implementation of algorithms to perform a fast analysis for checking compliance when the changes in the configuration are small and incremental, using results from the comprehensive exhaustive analysis of the configuration before the changes. Furthermore, studies can be made to show how the gap between the performance of the incremental-change analysis and the full exhaustive analysis narrows as the changes to the configuration increase in number and spatial distribution.

10.2.2 Statistical Analysis

In Chapter 6, we provided the formal framework that includes the measure (global compliance metric, and its various projections such as the number of violations and the fraction of possible traffic in violation) and the base statistical analysis algorithm that uses importance sampling. The biasing heuristic and stopping criteria parameterize the algorithm.

Following research directions can be explored to further extend the work on statistical analysis:

- A more fully-realized confidence measure computation mechanism, based on inverse Bayesian analysis, for the case where zero violations are reported when the stopping criteria is reached.
- Formalization of various classes of biasing heuristics and proofs of guaranteed rapid convergence, in the form of bounded relative error and/or asymptotic optimality, for each class of heuristics (e.g., graph-theoretic-distance-based, or user annotated and incremental configuration change based).
- Experimental proofs of accuracy and performance for different biasing heuristics, supplementing the analytic proofs when present. Our plan is to use the random topology and rule-sets generated by the mechanism mentioned earlier in this section for the experimental results.

APPENDIX A

TOPOLOGY INFERENCE DETAILS

A.1 Goal

The goal is to infer the underlying network topology from a variety of indirect information sources, such as the native configuration files for various Layer 3 devices, outputting the topology in XML that conforms to NetAPT's topology schema.

Currently, the only input information sources considered are the configuration files for Cisco PIX (v 6.x and 7.x) and ASA (v 8.x) series of firewalls and network security devices, Linux iptables firewall, and the Checkpoint UTM firewall.

A.2 Top-Level Approach

At the highest level, the flowchart shown in Figure A.1 summarizes how the NetAPT functions, with the sections relevant to topology inference highlighted:

The first step is to obtain the required input. As shown in Figure A.1, the tool can take information about the location of various firewalls (their IP addresses) and ways to establish secure connections to them (login name and password for users with sufficient read privileges for establishing ssh connections), and download the latest configuration information from the devices in their native format (PIX OS v 6.x, 7.x, or 8.x in the current implementation).

The other option is for the users of NetAPT to use third-party tools to obtain the configuration files. For example, the RANCID (Really Awesome New Cisco confIg Differ) tool can be used to maintain a repository of the latest configuration files for Cisco network devices. The tool uses CVS to maintain a history of changes. It manages the process of logging into the devices, running various commands to get the configuration information, and saving the said information in the

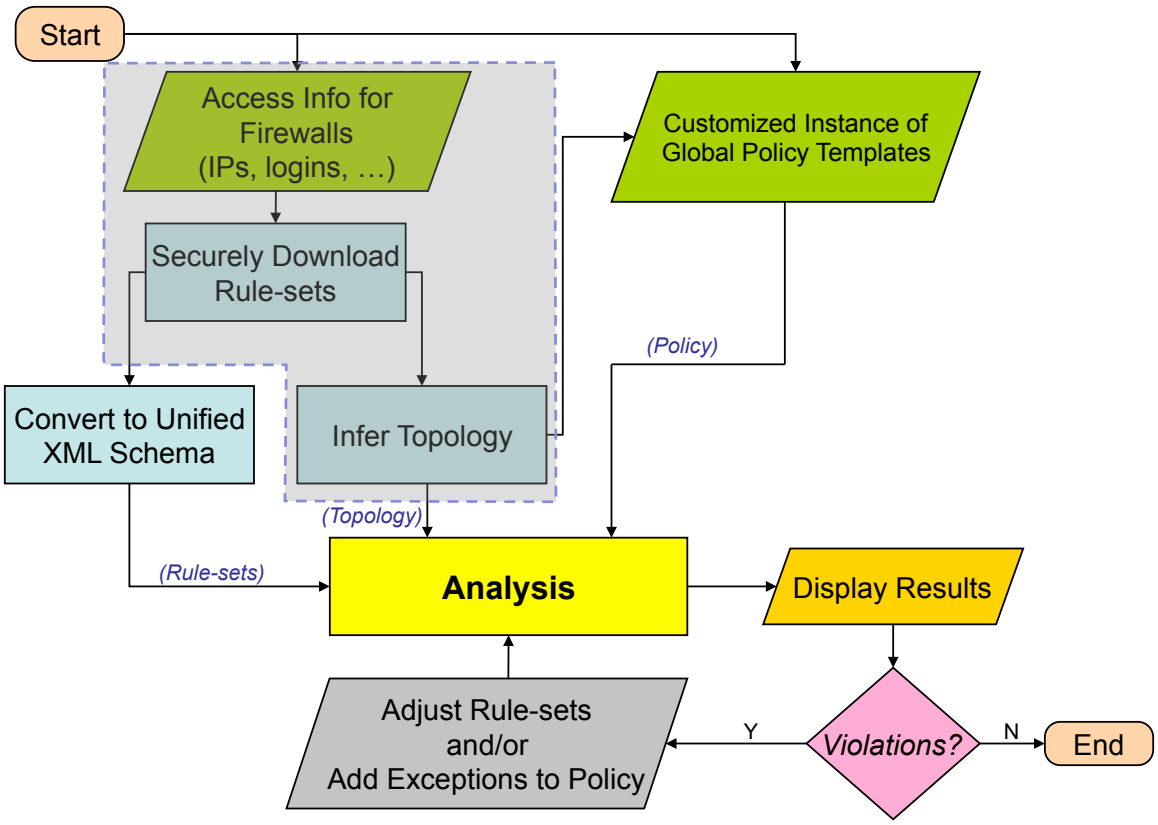

Figure A.1: Topology Inference

repository (duplicating a lot of NetAPT's functionality in this regard, except also using CVS). In such a scenario, NetAPT's engine simply needs to be invoked with the location of the repository on the local disk and it will proceed from there instead of trying to obtain the information itself.

A.3 Overall Topology Inference Framework

The main elements of the overall framework are:

1. *Topology database*: This database stores information about the myriad topology elements collected from different input sources. The input sources can potentially include firewall configuration files, dumps from nmap scans, and log files from routers and switches, to name a few. The framework includes scripts that provide the necessary interface to the database, while managing its internal integrity and conformance to the relational DB schema set out for it.

2. *Input scripts*: These scripts provide a modular and extensible way to parse the raw information from the various input sources and populate the topology database.

3. *Output scripts*: These scripts read the information contained in the topology database and produce "views" of the information in various formats that can be easily visualized and processed by other tools (or other components of the tool, in NetAPT's case). These formats include GraphML, Graphviz, and NetAPT's own topology XML schema.

A.3.1 Topology Database and Management Scripts

This functionality is currently provided using SQLite (http://www.sqlite.org/) for the relational DB and ANTFARM (http://antfarm.rubyforge.org/) for DB management. As a result, most the scripts are written in the Ruby scripting language and use the Ruby on Rails framework for DB integration.

A brief introduction to the Active Record terminology from Ruby on Rails is needed. An *Active Record* is an object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on the data. To summarize briefly:

1. *classes* typically represent/encapsulate *SQL tables*,

2. *objects* are the *rows* in the tables,

3. *base attributes* in each class are the *columns* in the corresponding tables, and

4. *association methods* in each class define *foreign keys* for the corresponding table (one-to-many, one-to-one, or many-to-one relations between rows of one table to rows of another table).

Furthermore, one can define methods that are to be performed before and immediately after an active record is created or updated, among other ways of manipulating them.

The database schema contains the following (relevant) tables/classes.

- ***Node***

This represents a node (i.e., a single device) in the topology; e.g., a firewall, a host, or a network switch. It has the following base attributes:

  - Name
  - Certainty_factor: the confidence of the inference algorithm that the node actually exists in the topology (a probability between 0 and 1).
  - Device_type: these are typically things like "firewall," "host," "switch," etc.
  - Custom: to store any specialized/extra information (currently unused).

  It has the following association methods (which define references to other classes):

  - *One-to-many associations*: layer2_interface, layer3_interface (via layer2_interface), service
  - *One-to-one associations*: operating_system

- ***Layer2_interface***

  This represents a layer2 (i.e., MAC) network interface. Base attributes:

  - Node_id: reference to the node to which this interface belongs.
  - Certainty_factor
  - Media_type: e.g., Ethernet or 802.11b
  - Custom

  Associations:

  - *One-to-many*: layer3_interface
  - *One-to-one*: ethernet_interface
  - *Belongs-to (Many-to-one)*: node

- ***Ethernet_interface***

  Base attributes:

- Address
- Custom

Associations:

- *Belongs-to (One-to-one)*: layer2_interface

- ***Layer3_interface***

  This represents a Layer 3 network interface (e.g., an IP address associated with a NIC). In general, though, any Layer 3 network protocol (may or may not be IP) can be used.

  Base attributes:

  - Certainty_factor
  - Protocol: Layer 3 network protocol being used (e.g., IP)
  - Custom

  Associations:

  - *One-to-one*: ip_interface
  - *Belongs-to (many-to-one)*: layer2_interface, layer3_network

- ***Ip_interface***

  This represents an IP address bound to a NIC. Base attributes:

  - Address: the IP address associated with this interface.
  - Virtual: a Boolean variable that indicates whether the IP address is associated with a virtual host in a firewall (used for static NAT mappings) rather than an actual physical device.
  - Custom

  Associations:

- *Belongs-to (one-to-one)*: layer3_interface

- ***Layer3_network***

  This represents a collection of layer 3 interfaces that form a single LAN. Base attributes:

  - Certainty_factor
  - Protocol
  - Custom

  Associations

  - *One-to-many*: layer3_interface
  - *One-to-one*: ip_network

- ***Ip_network***

  This represents a set of IP interfaces that form a single LAN (i.e., an IP-based Layer 3 network). Base attributes:

  - Address: CIDR-based address notation (network address/prefix length, e.g., 192.168.0.0/16)
  - Private: a Boolean variable that indicates whether the network is private (i.e., a subset of one of the following IP address spaces is being used: 10.0.0.0/8, 192.168.0.0/16, or 172.16.0.0/16).
  - Private_network_id: used to distinguish among multiple private networks with the same address
  - Custom

  Associations:

  - *Belongs-to (one-to-one)*: layer3_network

The attributes (both base and associated) can typically be accessed in an object-oriented fashion, e.g., node_obj.layer3_interfaces would return a list of all the layer3_interfaces objects that are associated with the node represented by node_obj.

Note that ANTFARM bundles each class mentioned above with a variety of methods for their manipulation, including those that are to be performed before and immediately after an active record object is created or updated. The ones that are relevant to our algorithm and/or were overwritten by us are described in the Input scripts section below.

A.3.2 Input Scripts

This section describes the input scripts used to parse configuration information from multiple Cisco PIX/ASA devices and populate the database described above.

Topology Elements

We assume the network topology to contain the following elements:

1. *Hosts*: devices typically with one interface (and associated IP address), though we support hosts with multiple interfaces.

2. *Networks*: a collection of hosts, representing a LAN, such that the constituent hosts can talk to each other without the need for Layer 3 routing.

3. *Firewalls*: devices with multiple (at least 2) interfaces, each connected to a network.

4. *Switches*: devices connected to multiple (at least 2) networks, and able to route traffic between any pair (typically represents a Layer 3 router). For switches, unlike firewalls, we may have information about only some of the associated interfaces. (Typically, a switch should have one interface for each connected network.)

A network is "primary" with respect to a firewall if it is directly connected to it, and "secondary" if it is connected to the firewall via a series of switches.

"Custom" Attribute

Our inference algorithms use the custom attributes associated with the various DB tables for a variety of purposes. The following is a description of what the strings assigned to that field mean in different contexts.

1. ***Ip_interface***

   (a) If the interface belongs to a *firewall*, it stores the *description string associated with that interface*, if any. When defining the details of an interface in PIX (v7.x) and ASA devices, one can also provide a description string elaborating the purpose of the interface. Our algorithm assumes that the description strings provide details about the network to which the interface is connected, and thus can be used in determining whether networks with similar addresses connected to different firewalls are actually one and the same, and thus can be *merged* and stored as one network in the database.

   (b) If the interface belongs to a *switch (router)*, it stores the list of addresses (CIDR) of the networks switched/routed via that device. (Addresses are stored as a ":" delimited string.)

   (c) For all other contexts (i.e., hosts), the field is empty.

2. ***Ip_network***

   - Names of the firewalls to which this network is connected (as primary or secondary). Again, stored as a ":" delimited string.

3. ***Layer3_network***

   - List of the firewall interface description strings for all the firewalls connected to this network (as primary or secondary, i.e., connected via a path that may include switches, but not any other firewalls).

Relevant Active Record Manipulation Functions

These are some of the functions that are relevant to the description of the main algorithm. The format is <return type> <function name> (<parameters>).

CLASS LAYER3_NETWORK

```
static Boolean desc_strings_intersect(String str1, String str2)
```

This function takes as input two sets of strings, each represented as a ":" delimited string. For example, the set {firewall1, firewall2} would berepresented as the string "firewall1:firewall2", and returns true if the two sets intersect. Note that this is a static/class function and not a member function.

```
static String desc_strings_union(String str1, String str2)
```

This function takes as input two sets of strings, each represented as a ":" delimited string, and returns their set union, also represented as a ":" delimited string.

```
static Layer3_network network_containing(String ip_net_cidr,
       String firewall_names_string)
```

This function takes as input the address of a network (as a CIDR string) and a list of firewall names to which the network is connected. It returns the database-stored network for which the argument is a sub-network, and is also connected to one of the firewalls to which this network is connected.

```
ip_networks = List of all ip_network objects stored in the database
for each ip_network object, ip_net, in ip_networks
        if ip_net_cidr is a subset of ip_net and
   desc_strings_intersect(firewall_names_string, ip_net.custom)
ip_net.custom = desc_strings_union(firewall_names_string,
                ip_net.custom)
update ip_net in the DB
return the layer3_network associated with the ip_network
       ip_net
else return nil
```

```
static Layer3_network network_addressed_exact(String ip_net_cidr,
       String firewall_names_string)
```

Similar to network_containing() above, except that it returns an exact match (if present) for argument network, and not a strict superset.

```
static Void merge(Layer3_network l3n)
```

Merges any of the networks currently contained in the database that ought to be subsumed by the argument network into the said network. Note that this is a static/class function and not a member function.

```
sub_networks = List of all layer3_networks in the DB that are
               subsets l3n (simply based on network address)
for each sub_network in sub_networks
        if sub_network != l3n and
           desc_strings_intersect(sub_network.custom, l3n.custom)
           description string of one of the firewalls connected
```

to each of the networks has the same description for the network.

```
Move all layer3_interface objects contained within sub_network to l3n
l3n.ip_network.custom = desc_strings_union
                        (l3n.ip_network.custom,
                         sub_network.ip_network.custom)
l3n.custom = desc_strings_union (l3n.custom,
             sub_network.custom)

update l3n in the database
destroy sub_network also destroys the corresponding
                    sub_newtork.ip_network object
```

CLASS IP_INTERFACE

Void create_layer3_interface()

Called whenever a new Ip_interface is created. Creates a new layer3_interface record associated with (has a one-to-one mapping with) this ip_interface record.
Among other things, calls create_ip_network() and assigns the return value as the layer3_network associated with the newly created layer3_interface.

Layer3_network create_ip_network()

Called whenever a new ip_interface is created. It checks to see whether a network already exists that should contain this interface. If one does not, it creates a small one that does. It only looks

```
for the networks that are connected to the firewall whose config is
currently being parsed.
l3n = Layer3_network.network_containing(address of this interface,
      name of the firewall currently being parsed)

if l3n is nil
        create a small, /29, network containing the address of this
      interface, and assign it to l3n

return l3n
```

fw_hostname	hostname of the firewall being parsed
pix_version	PIX OS main version number (6, 7, or 8)
ip_sec_peers	list of ipsec peers for VPN tunnels, obtained from the crypto map command
interfaces_wo_desc	list of the addresses (and the network mask of the connected network) of the firewall interfaces that have no description strings
interfaces_w_desc	list of the addresses (and network mask of the connected networks) of the firewall interfaces that also have a description string
description_strings	list of the description strings for the interfaces in "interfaces_w_desc" list, in the same order as that list
static_nat_actual	list of local/actual IP addresses from static NAT mappings (from static commands)
static_nat_virtual	list of global/virtual IP addresses from static NAT mappings (from static commands)
net_obj_ips	list of host IP addresses from network object host commands (object group definitions)
access_list_ips	list of addresses (and network masks) obtained from access-list commands (either as sources or destinations in various rules)
host_alias_ips	list of IP addresses obtained from the name command that defines string aliases for IPs
host_alias_map	a hash that maps aliases to IPs, as defined by the name command
gateway_ips	list of gateways/switches (IPs) obtained from route
routed_networks_map	a hash that maps each gateway IP to an array containing the network address and mask of each network that has that gateway in a route command

Table A.1: Lists and Strings Created During Input Parsing

Main input-parsing algorithm

The script takes as input the list of the paths to each of the firewall config files, or the path to the directory where the files are stored. It then processes each of the files in succession by calling the following function. This is a fairly high-level description. Some of the functionality may be summarized in italicized comments rather than pseudo-code. The Cisco PIX OS commands are italicized. (Full set of commands is available at [61].)

```
Parse(String path_to_file)
```

```
file = open(path_to_file, read)
```

```
Define various regular expressions for matching different
commands in the PIX OS command-set.

For each line in file, match the regular expressions and
create the lists and strings in Table A.1 (some lists
require matching multiple regexps over multiple lines)

Care is taken to dereference name aliases when the occur in
place of IP addresses in route, static, access-list and other
commands.

close(file)


Process firewall interfaces
for each ip_address in interfaces_wo_desc
        ip_if = new ip_interface object with the address ip_address

        put ip_if in the node object named fw_hostname (create the node
        if it doesnt already exist) and set its device_type as FW

        save ip_if in the DB
        Note that this will cause a call to
        ip_if.create_layer3_interface -> ip_if.create_ip_network
        as described above in the section on manipulation functions.
        As a result, the network connected to the firewall interface
        will be created and stored in the database as well.

for each ip_address in interfaces_w_desc
```

```
        description_str = corresponding entry in description_strings

        ip_if = new ip_interface object with the address ip_address

        put ip_if in the node object named fw_hostname (create the node
        if it doesnt already exist) and set its device_type as "FW"

        ip_if.custom = description_str

        save ip_if in the DB
        Again, corresponding networks (ip_network and layer3_network)
        are created

        update ip_if.layer3.interface.layer3_network.custom to
        include description_str
        The custom field of the layer3_network containing this
        interface now includes the description associated with this
        interface

        Layer3_network.merge(ip_if.layer3_interface.layer3_network)
        This merges all the networks in the DB that can be merged into
        the newly created layer3_network, i.e., all those networks whose
        address is a subset and at least one connected firewall had a
        matching description to one of this networks description set.
        See details of the merge function above.

Process network object groups
non_gateway_hosts = net_obj_ips  gateway_ips /* set subtration */
for each ip_address in non_gateway_ips
        ip_if = new ip_interface object with the address ip_address
```

```
        put ip_if in the node object named ip_address (create the node
        if it doesnt already exist) and set its device_type as "HOST"

        save ip_if in DB
        This will cause the new ip_interface (and the corresponding
        layer3_interface) to be added to the appropriate network
        connected to the current firewall. See class ip_interface
        manipulation functions above. This happens on all saves below.


Process IPs from name aliases, static commands and access-list
commands
other_ips = (host_alias_ips U static_nat_actual U access_list_ips)
            (net_obj_ips U gateway_ips)
for each ip_address in other_ips
ip_if = new ip_interface object with the address ip_address

        put ip_if in the node object named ip_address (create the node
        if it doesnt already exist)

        if an ip_interface record with the address ip_address doesnt
        already exist in the DB, save ip_if in the DB

Mark virtual IPs from static commands
for each ip_address in static_nat_virtual
        ip_if = new ip_interface object with the address ip_address
        ip_if.virtual = true

        put ip_if in the node object named ip_address (create the node
```

```
        if it doesnt already exist)

        if an ip_interface record with the address ip_address doesnt
        already exist in the DB, save ip_if in the DB
        else set the virtual attribute of the existing ip_interface
        record as true

Process gateways/switches
gateway_ips = gateway_ips  ip_sec_peers

for each ip_address in gateway_ips
        ip_if = new ip_interface object with the address ip_address

        put ip_if in the node object named ip_address@fw_hostname (create
        the node if it doesnt already exist) and set its device_type as
        "SWITCH"

        ip_if.custom = : delimited string containing list of network
        addresses contained in routed_networks_map[ip_address]

        save ip_if in the DB

        Now we create ip_network records for each routed network for
        this gateway
        for each network_addr in routed_networks_map[ip_address]
                l3_n = Layer3_network.network_addressed_exact(
                         network_addr, fw_hostname)

                if l3_n is nil
                        ip_n = new ip_network object with the address
```

```
        network_addr
        ip_n.custom = fw_hostname
        save ip_n in the DB
        This will also create the corresponding
        layer3_network record

l3_n = ip_n.layer3_network

l3_n.custom = Layer3_network.desc_string_union(
        l3_n.custom,
        ip_if.layer3_interface.layer3_network.custom)
ip_if.layer3_interface_layer_newtork.custom = l3_n.custom
Weve updated the l3_n custom field to include
descriptions for the primary network that contains
the gateway (and vice versa). This way descriptions
carry-over from primary networks to the secondary
networks connected to them.

save l3_n in the DB
update ip_if.layer3_interface_layer_newtork in the DB
```

A.3.3 Output Scripts

The output scripts access the topology database populated over potentially multiple runs of the input scripts and produce an XML description of the topology conforming to NetAPT's topology XML schema.

Main output-generation algorithm

The script takes as input the path to the location where the generated topology XML file would be stored (file extension .topo is used for these files). It can work in two modes: generate a single topology file summarizing all the information contained in the DB, or generate one file per connected component (in the graph-theoretic sense). In the latter case, the output file name specified is used as the prefix and each connected component is a self-contained topology XML file with names <prefix>_1.topo, <prefix>_2.topo, and so on.

The following are some of the lists that we use output generation algorithm.

- all_networks: list of all Layer 3 networks in the DB
- all_primary_networks: list of all networks directly connected to a firewall
- all_routed_networks: list of all networks connected to router/switch
- all_connected_networks: list of all networks reachable from a firewall (directly or via one or more switches)
- empty_nws_fws: map of addresses of the interfaces of firewalls connected to a network; only primary networks with no hosts (or switches) have keys in the map
- empty_routed_networks: list of networks connected to a switch (but not the switch's base network) with no hosts
- all_hosts: list of all hosts to be included in topology file
- all_switches: list of all switches to be included in the topology file
- all_firewalls: list of all firewalls

The steps of the algorithm are as follows.

Step 1: Generate list of all Layer 3 networks in the DB and various hash-maps for quick access to their attributes.

Remember that there's a one-to-one mapping between Layer3_network and ip_network tables/records.

```
for each l3_nw in DB.layer3_networks
add l3_nw to all_networks

store l3_nws details in hash-maps for quick access without
needing to refer to the DB repeatedly
For compactness of notation, from now on well refer to
details of layer3_network objects as simple member reference,
e.g., if l3_nw is such an object, its address and mask can be
accessed as l3_nw.address and l3_nw.netmask respectively. In
actuality though, it is more like networks_address_map[l3_nw]
etc.
```

Step 2: Generate a map of networks routed by each switch in the DB (a list of layer3_networks mapped to each switch's name) and lists of primary, routed, and connected networks

---

```
for each nd in DB.nodes
   if nd.device_type == SWITCH
       l3_if = node nds first layer3_interface (typically its only
           interface that we have information about)
       l3_nw = layer3_network that l3_if belongs to
           (l3_if.layer3_network)

       set the switchs name, sw_name, as nd.id (nds unique DB id),
       add sw_name to all_switches and
       store the switchs details in hash-maps
       hereby referred to as sw_name.<attribute>, but actually
       meaning switch_address_list[sw_name]; in particular,
       sw_name.base_network = l3_nw, i.e., the network the known
       switch interface belongs to
```

```
    add l3_nw to all_routed_networks, if not already present

    routed_networks = list of networks routed by this switch obtained
        from l3_if.ip_interface.custom string (note that this is
        a list of CIDR network addresses)
    fw_names_str = l3_nw.ip_network.custom
    This string is a : delimited list of firewalls connected to
    the switchs base network
    for each rtd_nw_addr in routed_networks
        rtd_l3_nw = layer3_network.network_addressed_exact(
            rtd_nw_addr, fw_names_str)
        This is now the appropriate layer3_network object/record

        if rtd_l3_nw != l3_nw and rtd_l3_nw ? sw_name.routed_networks
            add rtd_l3_nw to sw_name.routed_networks
            add rtd_l3_nw to all_routed_networks, if not present
            already
            add rtd_l3_nw to empty_routed_networks, if not present
            already
            add sw_name to rtd_l3_nw.switches


    if sw_name.routed_networks ∉ {}
    the route command in the firewall file didnt have the
    switch routing just to its base network (l3_nw)
        add sw_name to l3_nw.switches

if nd.device_type == FW
   for each l3_if in nd.layer3_interfaces
```

```
   if l3_if.layer3_network ∉ all_primary_networks
      add l3_if.layer3_network to all_primary_networks
          set empty_nws_fw[l3_if.layer3_network] as an empty list
```

Step 3: Generate the list of networks that will be included as <Network> elements in the generated topology. Initialize various lists and arrays for storing information about those networks, and add counter suffixes to the names of (different) networks that have the same address/mask.

```
all_connected_networks = all_primary_networks ? all_routed_networks

for each network in all_connected_networks
   if the map network_name_counter has the key network.cidr_addr
   this network was already encountered
      name_counter = network_name_counter[network.cidr_addr]
      network_name = network.cidr_addr +  - # + name_counter
      network_name_counter[network.cidr_addr] = name_counter + 1
   else
      network_name = network.cidr_addr
      network_name_counter[network.cidr_addr] = 2
   network_ids[network_name] = network
   network_name[network] = network_name
   hereby referred as network.name

   Initialize various other hash-maps that store information
   indexed by network_name; hereby referred to as
   network_name.firewalls, network_name.hosts and
   network_name.switches etc.
```

Step 4: Generate list of hosts in the DB, and update lists and containing information about primary and routed networks with no hosts. We are presuming that each host only has a single NIC.

```
for each nd in DB.nodes
   for each l3_if in nd.layer3_interfaces
      if (l3_if.layer3_network ? all_connected_networks and
         l3_if.ip_interface.virtual == false)

         if nd.device_type != FW and nd.device != SWITCH
         node is a host
            set host_id as l3_if and store details about the host in
            various maps indexed by host_id (e.g, host_id.address)

            add host_id to l3_if.layer3_network.name.hosts and
            to all_hosts

       if nd.device_type != FW
            remove the entry for the index l3_if.layer3_network from
            the map empty_nws_fw

            remove l3_if.layer3_network from the list
            empty_routed_networks

   else
if empty_nws_fw contains the key l3_if.layer3_network
   add l3_if.ip_interface.address to
           empty_nws_fw[l3_if.layer3_network]
```

Step 5: Generate list of dummy hosts for empty connected networks.

For NetAPT's analysis to consider traffic ending or beginning in a network, it needs to have at least one host.

```
Dummy hosts for primary networks
Assumption: less than 255 firewalls connected to each network
for each network in empty_nws_fw.keys
   address_prefix = first three bytes of network.address
   last_addr_byte = last byte of network.address

   fw_last_bytes = sorted list of last bytes of the addresses in
   the list empty_nws_fw[network]

   we now find the first gap in the entries of fw_last_bytes
   if fw_last_bytes.first > last_add_byte + 1
      host_last_byte = last_addr_byte + 1
   else
      for each fw_last_byte in fw_last_bytes
         if fw_last_byte + 1 ∉ fw_last_bytes
           host_last_byte = fw_last_byte + 1
   break_loop

   global_DH_index++

   host_address = address_prefix + host_last_byte
   host_id = DH # + global_DH_index + "(" + host_address + ")"

   add host_id to network.name.hosts and store details like
```

```
   host_address in maps indexed with host_id

   add host_id to all_hosts

   remove network from the list empty_routed_networks

Dummy hosts for empty routed networks
for each network in empty_routed_networks
   address_prefix = first three bytes of network.address
   last_addr_byte = last byte of network.address
   host_last_byte = last_addr_byte + 1

   global_DH_index++

   host_address = address_prefix + host_last_byte
   host_id = DH # + global_DH_index + "(" + host_address + ")"

   add host_id to network.name.hosts and store details like
   host_address in maps indexed with host_id

   add host_id to all_hosts
```

Step 6: Generate the list of firewalls.

```
for each nd in DB.nodes
   nd_processed_flag = false
   initialize fw_id
   for each l3_if in nd.layer3_interfaces
```

```
        if nd_processed_flag == false
            if nd.device_type == "FW"
        fw_id = nd.name
        fw_id.name = "Firewall " + fw_id
            nd_processed_flag = true

    add l3_if.ip_interface.address to fw_id.addresses and
        l3_if.layer3_network to fw_id.connected_networks and
        fw_id to l3_if.layer3_network.name.firewalls
```

Step 7-1: Generate a single topology file.

This step is executed if the script was invoked indicating that only a single output file is to be generated.

```
topo_file = open(file_name, "w")

output APT XML DTD to topo_file

Add <Network> elements
for each network in all_connected_networks
   Output details from the various network hash-maps, such as
   network.address, network.mask and network.name

Add <Host> elements
for each host in all_hosts
   Output details from the various host hash-maps

Add <Switch> elements
```

```
for each switch in all_switches
   Output details from the various switch hash-maps, including
   switch.routed_networks and switch.base_network

Add <Firewall> elements
for each firewall in all_firewalls
   Output details from the various firewall hash-maps, including
   firewall.addresses and firewall.networks

close topo_file
```

Step 7-2: Generate multiple topology files.

This step is executed if the script was invoked with the option of generating one file per (graph-theoretic) connected component.

```
global_visited_firewalls = {}
file_counter = 1

for each firewall in all_firewalls
   if firewall ∉ global_visited_firewalls
       local_file_name = file_name + "_" + file_counter
       file_counter++
       topo_file = open(local_file_name, "w")

       Initialize local lists for elements to include in this
       connected component
       local_hosts = {}
       local_networks = {}
       local_firewalls = {}
```

```
Initialize various lists to keep track of visited elements
local_visited_switches = {}
local_to_visit_switches ={}
local_to_visit_firewalls = {}

push: adding to the end of the list, pop: removing (and
returning) from the front of the list
local_to_visit_firewalls.push(firewall)
local_firewalls.push(firewall)

while local_to_visit_firewalls ≠ {}
    current_fw = local_to_visit_firewall.front

    for each nw_name ∈ current_fw.networks
        if nw_name ∉ local_networks
            local_networks.push(nw_name)
            local_hosts = local_hosts ∪ nw_name.hosts
            local_to_visit_switches = (local_to_visit_switches ∪
                nw_name.switches)  local_visited_switches

            for each fw_to_visit in nw_name.firewalls
                if (fw_to_visit ≠ current_fw and
                    fw_to_visit ∉ global_visited_firewalls)
                    local_to_visit_firewalls.push(fw_to_visit)

    while local_to_visit_switches ≠ {}
        current_sw = local_to_visit_switches.pop
        local_visited_switches.push(current_sw)
        routed_networks = current_sw.routed_networks ∪
```

```
                {current_sw.base_network}

            for each routed_network in routed_networks
                nw_name = routed_network.name
                if nw_name ∉ local_networks
                    local_networks.push(nw_name)
                    local_hosts = local_hosts ∪ nw_name.hosts
                    local_to_visit_switches = (local_to_visit_switches ∪
                        nw_name.switches)  local_visited_switches

        for each fw_to_visit in nw_name.firewalls
                    if (fw_to_visit ≠ current_fw and
                        fw_to_visit ∉ global_visited_firewalls)
                        local_to_visit_firewalls.push(fw_to_visit)

        visited_firewalls.push(current_fw)
        local_firewalls = local_firewalls ∪ currrent_fw
        local_to_visit_firewalls.pop

output APT XML DTD to topo_file

Add <Network> elements
for each network in all_connected_networks
   Output details from the various network hash-maps, such as
   network.address, network.mask and network.name

Add <Host> elements
for each host in all_hosts
   Output details from the various host hash-maps
```

```
Add <Switch> elements
for each switch in all_switches
   Output details from the various switch hash-maps, including
   switch.routed_networks and switch.base_network

Add <Firewall> elements
for each firewall in all_firewalls
   Output details from the various firewall hash-maps, including
   firewall.addresses and firewall.networks

close topo_file
```

REFERENCES

[1] SANS Institute, "SANS: The Top Cyber Security Risks," http://www.sans.org/top-cyber-security-risks/, accessed April 2012.

[2] A. Wool, "A Quantitative Study of Firewall Configuration Errors," *Computer*, vol. 37, no. 6, pp. 62–67, June 2004.

[3] Netfilter.org, "The Netfilter.org 'iptables' Project," http://www.netfilter.org/projects/iptables/index.html, 2010.

[4] 3Com Corporation, "3Com Embedded Firewall Solution," http://www.sans.org/reading_room/whitepapers/firewalls/3com-distributed-embedded-firewall_1435, April 2004.

[5] Centre for the Protection of National Infrastructure, "Firewall Deployment for SCADA and Process Control Networks: Good Practice Guide," http://energy.gov/sites/prod/files/Good Practices Guide for Firewall Deployment.pdf, February 2005.

[6] National Security Agency, "Security-Enhanced Linux," http://www.nsa.gov/research/selinux/, January 2009.

[7] Cisco Systems, "Cisco Security Agent," http://www.cisco.com/en/US/products/sw/secursw/ps5057/index.html, accessed April 2012.

[8] Trusted Computing Group, "Network Security," http://www.trustedcomputinggroup.org/solutions/network_security, November 2011.

[9] Cisco Systems, "Release Notes for Cisco Clean Access (NAC Appliance) Version 3.6(4)," http://www.cisco.com/en/US/docs/security/nac/appliance/release_notes/36/36rn.html, April 2012.

[10] Team Cymru, "The Bogon Reference," http://www.team-cymru.org/Services/Bogons/, 2012.

[11] W. R. Cheswick, S. M. Bellovin, and A. D. Rubin, *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison-Wesley, 2003.

[12] M. G. Gouda and A. X. Liu, "Complete Redundancy in Detection in Firewalls," in *Proceedings of the 19th Annual IFIP Conference on Data and Applications Security*, 2005, pp. 196–209.

[13] M. G. Gouda and A. X. Liu, "Firewall Design: Consistency, Completeness, and Compactness," in *Proceeding of the 2004 International Conference on Dependable Systems and Networks (DSN 2004)*, Tokyo, Japan, March 2004, pp. 320–327.

[14] A. Hari, S. Suri, and G. M. Parulkar, "Detecting and Resolving Packet Filter Conflicts," in *Proceedings of 19th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, vol. 3, Tel Aviv, Israel, March 2000, pp. 1203–1212.

[15] D. Eppstein and S. Muthukrishnan, "Internet Packet Filter Management and Rectangle Geometry," in *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA 2001)*. Society for Industrial and Applied Mathematics, 2001, pp. 827–835.

[16] F. Boboescu and G. Varghese, "Fast and Scalable Conflict Detection for Packet Classifiers," *Computer Networks*, vol. 42, no. 6, pp. 717–735, 2003.

[17] E. Al-Shaer and H. Hamed, "Firewall Policy Advisor for Anomaly Detection and Rule Editing," in *Proceedings of the IEEE/IFIP 8th International Symposium on Integrated Network Management (IM 2003)*, Colorado Springs, CO, March 2003, pp. 17–30.

[18] E. Al-Shaer and H. Hamed, "Management and Translation of Filtering Security Policies," in *Proceedings of the 38th IEEE International Conference on Communications (ICC 2003)*, Anchorage, AK, May 2003, pp. 256–260.

[19] E. Al-Shaer and H. Hamed, "Discovery of Policy Anomalies in Distributed Firewalls," in *Proceedings of the 23rd Conference of the IEEE Communications Society (INFOCOM 2004)*, vol. 4, Hong Kong, March 2004, pp. 2605–2616.

[20] L. Yuan, J. Mai, Z. Shu, H. Chen, C. Chuah, and P. Mohapatra, "FIREMAN: A Toolkit for Firewall Modeling and Analysis," in *Proceedings of IEEE Symposium on Security and Privacy*, Oakland, CA, 2006, pp. 199–213.

[21] J. Hwang, T. Xie, V. Hu, and M. Altunay, "Mining Likely Properties of Access Control Policies via Association Rule Mining," in *Proceedings of the 24th Annual IFIP WG 11.3 Working Conference on Data and Applications Security (DBSec 2010)*, June 2010, pp. 193–208.

[22] V. Hu, R. Kuhn, T. Xie, and J. Hwang, "Model Checking for Verification of Mandatory Access Control Models and Properties," *International Journal of Software Engineering and Knowledge Engineering*, vol. 21, no. 1, pp. 103–127, 2011.

[23] A. X. Liu, F. Chen, J. Hwang, and T. Xie, "Designing Fast and Scalable XACML Policy Evaluation Engines," *IEEE Transactions on Computers*, vol. 60, no. 12, pp. 1802–1817, December 2011.

[24] F. Chen, A. X. Liu, J. Hwang, and T. Xie, "First Step Towards Automatic Correction of Firewall Policy Faults," in *Proceedings of the 24th USENIX Large Installation System Administration Conference (LISA 2010)*, November 2010, pp. 75–90.

[25] J. D. Guttman, "Filtering Postures: Local Enforcement for Global Policies," in *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 1997, pp. 120–129.

[26] Y. Bartal, A. Mayer, K. Nissim, and A. Wool, "Firmato: A Novel Firewall Management Toolkit," *ACM Transactions on Computer Systems*, vol. 22, no. 4, pp. 381–420, 2004.

[27] Cisco Systems, "CiscoWorks Management Center for Firewalls," http://www.cisco.com/en/US/products/sw/cscowork/ps3992/index.html, accessed April 2008.

[28] E. Clark, O. Grumberg, and D. Peled, *Model Checking*. Cambridge, MA: The MIT Press, January 2000.

[29] R. W. Ritchey and P. Ammann, "Using Model Checking to Analyze Network Vulnerabilities," in *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, 2000, pp. 156–165.

[30] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. M. Wing, "Automated Generation and Analysis of Attack Graphs," in *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, 2002, pp. 273–284.

[31] X. Ou, S. Govindavajhala, and A. W. Appel, "MulVAL: A Logic-Based Network Security Analyzer," in *Proceedings of the 14th USENIX Security Symposium*, Baltimore, MD, 2005, pp. 113–128.

[32] X. Ou, W. F. Boyer, and M. A. McQueen, "A Scalable Approach to Attack Graph Generation," in *Proceedings of 13th ACM Conference on Computer and Communications Security (CCS 2006)*, Alexandria, VA, 2006, pp. 336–345.

[33] S. Singh, J. Lyons, and D. M. Nicol, "Fast Model-Based Penetration Testing," in *Proceedings of 2004 Winter Simulation Conference (WSC'04)*, Washington, DC, December 2004, pp. 309–317.

[34] OASIS, "OASIS eXtensible Access Control Markup Language (XACML) TC," http://oasis-open.org/committees/xacml/, 2012.

[35] H. Yin, J. Zhou, H. Wu, and L. Yu, "A SAML/XACML Based Access Control between Portal and Web Services," in *Proceedings of First International Symposium on Data, Privacy and E-Commerce (ISDPE 2007)*, Chengdu, China, November 2007, pp. 356–360.

[36] G. Ahn and R. Sandhu, "Role-Based Authorization Constraints Specification," *ACM Transactions on Information and System Security (TISSEC)*, vol. 3, no. 4, pp. 207–226, November 2000.

[37] L. Lymberopoulos, E. Lupu, and M. Sloman, "An Adaptive Policy-Based Framework for Network Services Management," *Journal of Network and System Management*, vol. 11, no. 3, pp. 277–303, Sep. 2003.

[38] A. Pnueli, "The Temporal Logic of Programs," in *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS)*, 1977, pp. 46–57.

[39] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications," *ACM Transactions on Programming Languages and Systems*, vol. 8, no. 2, pp. 244–263, 1986.

[40] Skybox Security, "Skybox Firewall Assurance," http://www.skyboxsecurity.com/products/firewall-assurance/, accessed April 2012.

[41] RedSeal Networks, "RedSeal Network Advisor," http://www.redsealnetworks.com/products/networkadvisor/, accessed April 2012.

[42] Arbor Networks, "Peakflow X: Enterprise Network Security," http://www.arbornetworks.com/peakflow-x-enterprise-network-security.html, accessed April 2012.

[43] H. Edelsbrunner, "A New Approach to Rectangle Intersections, Part I," *International Journal of Computer Mathematics*, vol. 13, pp. 209–219, 1983.

[44] H. Edelsbrunner, "A New Approach to Rectangle Intersections, Part II," *International Journal of Computer Mathematics*, vol. 13, pp. 221–229, 1983.

[45] J. Hao and J. B. Orlin, "A Faster Algorithm for Finding the Minimum Cut in a Directed Graph," *Journal of Algorithms*, vol. 17, pp. 424–446, 1994.

[46] C. S. Chekuri, A. V. Goldberg, D. R. Karger, M. S. Levine, and C. Stein, "Experimental Study of Minimum Cut Algorithms," in *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 1997, pp. 324–333.

[47] P. Heidelberger, "Fast Simulation of Rare Events in Queueing and Reliability Models," *ACM Transactions on Modeling and Computer Simulation*, vol. 5, no. 1, pp. 43–85, 1995.

[48] P. Glasserman, P. Heidelberger, and P. Shahabuddin, "Asymptotically Optimal Importance Sampling and Stratification for Pricing Path-Dependent Options," *Mathematical Finance*, vol. 9, no. 2, pp. 117–152, April 1999.

[49] G. Biondini, W. L. Kath, and C. R. Menyuk, "Importance Sampling for Polarization-mode Dispersion," *IEEE Photonics Technology Letters*, vol. 14, no. 3, pp. 310–312, March 2002.

[50] M. Falkner, M. Devetsikiotis, and I. Lambadaris, "Fast Simulation of Networks of Queues with Effective and Decoupling Bandwidths," *ACM Transactions on Modeling and Computer Simulation*, vol. 9, no. 1, pp. 45–58, January 1999.

[51] U. S. Department of Energy Office of Electricity Delivery and Energy Reliability, "Advanced Network Toolkit for Assessments and Remote Mapping (ANTFARM)," http://energy.gov/sites/prod/files/oeprod/DocumentsandMedia/12-ANTFARM.pdf, August 2008.

[52] Shrubbery Networks, "RANCID: Really Awesome New Cisco config Differ," http://www.shrubbery.net/rancid/, accessed April 2012.

[53] JGraph Ltd, "Java Graph Drawing Component," http://www.jgraph.com/jgraph.html, accessed April 2012.

[54] J. O'Madadhain, D. Fisher, and T. Nelson, "JUNG - Java Universal Network/Graph Framework," http://jung.sourceforge.net/, accessed April 2012.

[55] K. Stouffer, J. Falco, and K. Scarfone, "Guide to Industrial Control Systems (ICS) Security," National Institute of Standards and Technology (NIST), Tech. Rep. SP-800-82 (Second Public Draft), September 2007.

[56] North American Electric Reliability Corporation, "Standard CIP-005-1: Cyber Security - Electronic Security Perimeter(s)," http://www.nerc.com/files/CIP-005-1.pdf, June 2006.

[57] D. J. Watts and S. H. Strogatz, "Collective Dynamics of 'Small World' Networks," *Nature*, vol. 393, no. 6684, pp. 409–410, 1998.

[58] G. K. Zipf, *Human Behavior and the Principle of Least Effort*. Cambridge, MA: Addison-Wesley, 1949.

[59] B. C. Arnold, *Pareto Distributions*. International Co-operative Publishing House, 1983.

[60] A.-L. Barbási and R. Albert, "Emergence of Scaling in Random Networks," *Science*, vol. 286, no. 5439, pp. 509–512, October 1999.

[61] Cisco Systems, "Cisco ASA 5580 Adaptive Security Appliance Command Reference, Version 8.1," http://www.cisco.com/en/US/docs/security/asa/asa81/command/ref/refgd.html, 2008.